Scott Rifenbark
Scotty's Documentation Services, INC
<srifenbark@gmail.com>

Manual Notes

- This version of the ***Yocto Project Application Development and the Extensible Software Development Kit (eSDK)*** manual is for the 2.5 release of the Yocto Project. To be sure you have the latest version of the manual for this release, go to the Yocto Project documentation page and select the manual from that site. Manuals from the site are more up-to-date than manuals derived from the Yocto Project released TAR files.
- If you located this manual through a web search, the version of the manual might not be the one you want (e.g. the search might have returned a manual much older than the Yocto Project version with which you are working). You can see all Yocto Project major releases by visiting the Releases page. If you need a version of this manual for a different Yocto Project release, visit the Yocto Project documentation page and select the manual set by using the "ACTIVE RELEASES DOCUMENTATION" or "DOCUMENTS ARCHIVE" pull-down menus.
- To report any inaccuracies or problems with this manual, send an email to the Yocto Project discussion group at `yocto@yoctoproject.com` or log into the freenode `#yocto` channel.

Revision History	
Revision 2.1	April 2016
Released with the Yocto Project 2.1 Release.	
Revision 2.2	October 2016
Released with the Yocto Project 2.2 Release.	
Revision 2.3	May 2017
Released with the Yocto Project 2.3 Release.	
Revision 2.4	October 2017
Released with the Yocto Project 2.4 Release.	
Revision 2.5	May 2018
Released with the Yocto Project 2.5 Release.	

Table of Contents

Chapter 1. Introduction¶

Table of Contents

1.1. Introduction¶

Welcome to the Yocto Project Application Development and the Extensible Software Development Kit (eSDK) manual. This manual provides information that explains how to use both the Yocto Project extensible and standard SDKs to develop applications and images. Additionally, the manual also provides information on how to use the popular Eclipse™ IDE as part of your application development workflow within the SDK environment.

> **Note**
> Prior to the 2.0 Release of the Yocto Project, application development was primarily accomplished through the use of the Application Development Toolkit (ADT) and the availability of stand-alone cross-development toolchains and other tools. With the 2.1 Release of the Yocto Project, application development has transitioned to within a tool-rich extensible SDK and the more traditional standard SDK.

All SDKs consist of the following:

- ***Cross-Development Toolchain***: This toolchain contains a compiler, debugger, and various miscellaneous tools.
- ***Libraries, Headers, and Symbols***: The libraries, headers, and symbols are specific to the image (i.e. they match the image).
- ***Environment Setup Script***: This `*.sh` file, once run, sets up the cross-development environment by defining variables and preparing for SDK use.

Additionally, an extensible SDK has tools that allow you to easily add new applications and libraries to an image, modify the source of an existing component, test changes on the target hardware, and easily integrate an application into the OpenEmbedded build system.

You can use an SDK to independently develop and test code that is destined to run on some target machine. SDKs are completely self-contained. The binaries are linked against their own copy of `libc`, which results in no dependencies on the target system. To achieve this, the pointer to the dynamic loader is configured at install time since that path cannot be dynamically altered. This is the reason for a wrapper around the `populate_sdk` and `populate_sdk_ext` archives.

Another feature for the SDKs is that only one set of cross-compiler toolchain binaries are produced for any given architecture. This feature takes advantage of the fact that the target hardware can be passed to `gcc` as a set of compiler options. Those options are set up by the environment script and contained in variables such as CC and LD. This reduces the space needed for the tools. Understand, however, that every target still needs a sysroot because those binaries are target-specific.

The SDK development environment consists of the following:

- The self-contained SDK, which is an architecture-specific cross-toolchain and matching sysroots (target and native) all built by the OpenEmbedded build system (e.g. the SDK). The toolchain and sysroots are based on a Metadata configuration and extensions, which allows you to cross-develop on the host machine for the target hardware. Additionally, the extensible SDK contains the `devtool` functionality.
- The Quick EMUlator (QEMU), which lets you simulate target hardware. QEMU is not literally part of the SDK. You must build and include this emulator separately. However, QEMU plays an important role in the development process that revolves around use of the SDK.
- The Eclipse IDE Yocto Plug-in. This plug-in is available for you if you are an Eclipse user. In the same manner as QEMU, the plug-in is not literally part of the SDK but is rather available for use as part of the development process.
- Various performance-related tools that can enhance your development experience. These tools are also separate from the actual SDK but can be independently obtained and used in the development process.

In summary, the extensible and standard SDK share many features. However, the extensible SDK has powerful development tools to help you more quickly develop applications. Following is a table that summarizes the primary differences between the standard and extensible SDK types when considering which to build:

Feature	***Standard SDK***	***Extensible SDK***
Toolchain	Yes	Yes*
Debugger	Yes	Yes*
Size	100+ MBytes	1+ GBytes (or 300+ MBytes for minimal w/toolchain)
`devtool`	No	Yes
Build Images	No	Yes
Updateable	No	Yes
Managed Sysroot**	No	Yes
Installed Packages	No***	Yes****
Construction	Packages	Shared State

* Extensible SDK contains the toolchain and debugger if SDK_EXT_TYPE is "full" or SDK_INCLUDE_TOC

** Sysroot is managed through the use of `devtool`. Thus, it is less likely that you will corrupt your SDK sys

*** You can add runtime package management to the standard SDK but it is not supported by default.

**** You must build and make the shared state available to extensible SDK users for "packages" you want to enable

1.1.1. The Cross-Development Toolchain¶

The Cross-Development Toolchain consists of a cross-compiler, cross-linker, and cross-debugger that are used to develop user-space applications for targeted hardware. Additionally, for an extensible SDK, the toolchain also has built-in `devtool` functionality. This toolchain is created by running a SDK installer script or through a Build Directory that is based on your metadata configuration or extension for your targeted device. The cross-toolchain works with a matching target sysroot.

1.1.2. Sysroots¶

The native and target sysroots contain needed headers and libraries for generating binaries that run on the target architecture. The target sysroot is based on the target root filesystem image that is built by the OpenEmbedded build system and uses the same metadata configuration used to build the cross-toolchain.

1.1.3. The QEMU Emulator¶

The QEMU emulator allows you to simulate your hardware while running your application or image. QEMU is not part of the SDK but is made available a number of different ways:

- If you have cloned the `poky` Git repository to create a Source Directory and you have sourced the environment setup script, QEMU is installed and automatically available.
- If you have downloaded a Yocto Project release and unpacked it to create a Source Directory and you have sourced the environment setup script, QEMU is installed and automatically available.
- If you have installed the cross-toolchain tarball and you have sourced the toolchain's setup environment script, QEMU is also installed and automatically available.

1.1.4. Eclipse™ Yocto Plug-in¶

The Eclipse IDE is a popular development environment and it fully supports development using the Yocto Project. When you install and configure the Eclipse Yocto Project Plug-in into the Eclipse IDE, you maximize your Yocto Project experience. Installing and configuring the Plug-in results in an environment that has extensions specifically designed to let you more easily develop software. These extensions allow for cross-compilation, deployment, and execution of your output into a QEMU emulation session. You can also perform cross-debugging and profiling. The environment also supports many performance-related tools that enhance your development experience.

> **Note**
> Previous releases of the Eclipse Yocto Plug-in supported "user-space tools" (i.e. LatencyTOP, PowerTOP, Perf, SystemTap, and Lttng-ust) that also added to the development experience. These tools have been deprecated with the release of the Eclipse Yocto Plug-in.

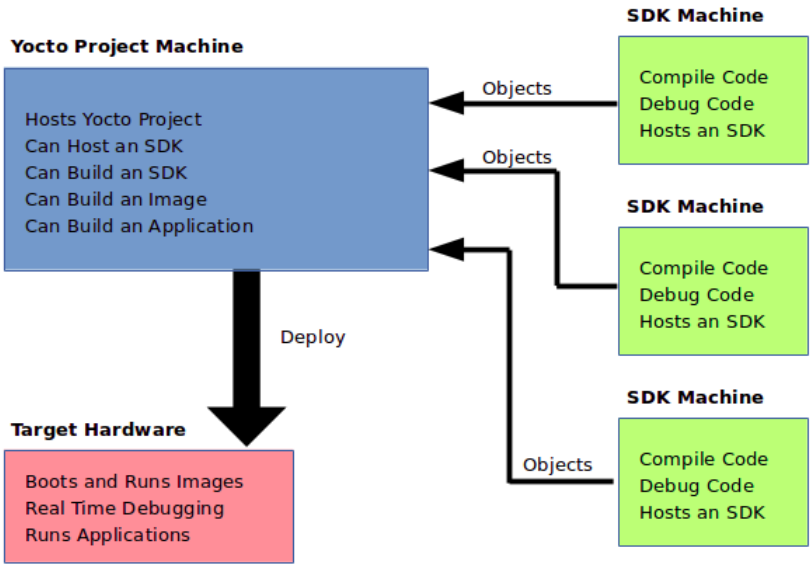
For information about the application development workflow that uses the Eclipse IDE and for a detailed example of how to install and configure the Eclipse Yocto Project Plug-in, see the "Developing Applications Using Eclipse™" Chapter.

1.1.5. Performance Enhancing Tools¶

Supported performance enhancing tools are available that let you profile, debug, and perform tracing on your projects developed using Eclipse. For information on these tools see http://www.eclipse.org/linuxtools/.

1.2. SDK Development Model¶

Fundamentally, the SDK fits into the development process as follows:



The SDK is installed on any machine and can be used to develop applications, images, and kernels. An SDK can even be used by a QA Engineer or Release Engineer. The fundamental concept is that the machine that has the SDK installed does not have to be associated with the machine that has the Yocto Project installed. A developer can independently compile and test an object on their machine and then, when the object is ready for integration into an image, they can simply make it available to the machine that has the Yocto Project. Once the object is available, the image can be rebuilt using the Yocto Project to produce the modified image.

You just need to follow these general steps:

1. ***Install the SDK for your target hardware:*** For information on how to install the SDK, see the "Installing the SDK" section.

2. ***Download or Build the Target Image:*** The Yocto Project supports several target architectures and has many pre-built kernel images and root filesystem images.

   If you are going to develop your application on hardware, go to the `machines` download area and choose a target machine area from which to download the kernel image and root filesystem. This download area could have several files in it that support development using actual hardware. For example, the area might contain `.hddimg` files that combine the kernel image with the filesystem, boot loaders, and so forth. Be sure to get the files you need for your particular development process.

   If you are going to develop your application and then run and test it using the QEMU emulator, go to the `machines/qemu` download area. From this area, go down into the directory for your target architecture (e.g. `qemux86_64` for an Intel®-based 64-bit architecture). Download the kernel, root filesystem, and any other files you need for your process.

   Note
   To use the root filesystem in QEMU, you need to extract it. See the "Extracting the Root Filesystem" section for information on how to extract the root filesystem.

3. ***Develop and Test your Application:*** At this point, you have the tools to develop your application. If you need to separately install and use the QEMU emulator, you can go to QEMU Home Page to download and learn about the emulator. See the "Using the Quick EMUlator (QEMU)" chapter in the Yocto Project Development Tasks Manual for information on using QEMU within the Yocto Project.

The remainder of this manual describes how to use the extensible and standard SDKs. Information also exists in appendix form that describes how you can build, install, and modify an SDK.

Chapter 2. Using the Extensible SDK¶

Table of Contents

This chapter describes the extensible SDK and how to install it. Information covers the pieces of the SDK, how to install it, and presents a look at using the devtool functionality. The extensible SDK makes it easy to add new applications and libraries to an image, modify the source for an existing component, test changes on the target hardware, and ease integration into the rest of the OpenEmbedded build system.

> **Note**
> For a side-by-side comparison of main features supported for an extensible SDK as compared to a standard SDK, see the "Introduction" section.

In addition to the functionality available through devtool, you can alternatively make use of the toolchain directly, for example from Makefile, Autotools, and Eclipse™-based projects. See the "Using the SDK Toolchain Directly" chapter for more information.

2.1. Why use the Extensible SDK and What is in It?¶

The extensible SDK provides a cross-development toolchain and libraries tailored to the contents of a specific image. You would use the Extensible SDK if you want a toolchain experience supplemented with the powerful set of devtool commands tailored for the Yocto Project environment.

The installed extensible SDK consists of several files and directories. Basically, it contains an SDK environment setup script, some configuration files, an internal build system, and the devtool functionality.

2.2. Installing the Extensible SDK¶

The first thing you need to do is install the SDK on your Build Host by running the *.sh installation script.

You can download a tarball installer, which includes the pre-built toolchain, the runqemu script, the internal build system, devtool, and support files from the appropriate toolchain directory within the Index of Releases. Toolchains are available for several 32-bit and 64-bit architectures with the x86_64 directories, respectively. The toolchains the Yocto Project provides are based off the core-image-sato and core-image-minimal images and contain libraries appropriate for developing against that image.

The names of the tarball installer scripts are such that a string representing the host system appears first in the filename and then is immediately followed by a string representing the target architecture. An extensible SDK has the string "-ext" as part of the name. Following is the general form:

```
poky-glibc-host_system-image_type-arch-toolchain-ext-release_version.sh

Where:
    host_system is a string representing your development system:

               i686 or x86_64.

    image_type is the image for which the SDK was built:

               core-image-sato or core-image-minimal

    arch is a string representing the tuned target architecture:

               aarch64, armv5e, core2-64, i586, mips32r2, mips64, ppc7400, or cortexa8hf-neon

    release_version is a string representing the release number of the Yocto Project:

               2.5, 2.5+snapshot
```

For example, the following SDK installer is for a 64-bit development host system and a i586-tuned target architecture based off the SDK for core-image-sato and using the current 2.5 snapshot:

```
poky-glibc-x86_64-core-image-sato-i586-toolchain-ext-2.5.sh
```

> **Note**
> As an alternative to downloading an SDK, you can build the SDK installer. For information on building the installer, see the "Building an SDK Installer" section. Another helpful resource for building an installer is the Cookbook guide to Making an Eclipse Debug Capable Image wiki page. This wiki page focuses on development when using the Eclipse IDE.

The SDK and toolchains are self-contained and by default are installed into the `poky_sdk` folder in your home directory. You can choose to install the extensible SDK in any location when you run the installer. However, because files need to be written under that directory during the normal course of operation, the location you choose for installation must be writable for whichever users need to use the SDK.

The following command shows how to run the installer given a toolchain tarball for a 64-bit x86 development host system and a 64-bit x86 target architecture. The example assumes the SDK installer is located in `~/Downloads/` and has execution rights.

> **Note**
> If you do not have write permissions for the directory into which you are installing the SDK, the installer notifies you and exits. For that case, set up the proper permissions in the directory and run the installer again.

```
$ ./Downloads/poky-glibc-x86_64-core-image-minimal-core2-64-toolchain-ext-2.5.sh
Poky (Yocto Project Reference Distro) Extensible SDK installer version 2.5
==========================================================================
Enter target directory for SDK (default: ~/poky_sdk):
You are about to install the SDK to "/home/scottrif/poky_sdk". Proceed[Y/n]? Y
Extracting SDK..............done
Setting it up...
Extracting buildtools...
Preparing build system...
Parsing recipes: 100% |#################################################################| Time: 0:00:52
Initialising tasks: 100% |##############################################################| Time: 0:00:00
Checking sstate mirror object availability: 100% |######################################| Time: 0:00:00
Loading cache: 100% |###################################################################| Time: 0:00:00
Initialising tasks: 100% |##############################################################| Time: 0:00:00
done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
 $ . /home/scottrif/poky_sdk/environment-setup-core2-64-poky-linux
```

2.3. Running the Extensible SDK Environment Setup Script¶

Once you have the SDK installed, you must run the SDK environment setup script before you can actually use the SDK. This setup script resides in the directory you chose when you installed the SDK, which is either the default `poky_sdk` directory or the directory you chose during installation.

Before running the script, be sure it is the one that matches the architecture for which you are developing. Environment setup scripts begin with the string "`environment-setup`" and include as part of their name the tuned target architecture. As an example, the following commands set the working directory to where the SDK was installed and then source the environment setup script. In this example, the setup script is for an IA-based target machine using i586 tuning:

```
$ cd /home/scottrif/poky_sdk
$ source environment-setup-core2-64-poky-linux
SDK environment now set up; additionally you may now run devtool to perform development tasks.
Run devtool --help for further details.
```

Running the setup script defines many environment variables needed in order to use the SDK (e.g. `PATH`, `CC`, `LD`, and so forth). If you want to see all the environment variables the script exports, examine the installation file itself.

2.4. Using `devtool` in Your SDK Workflow¶

The cornerstone of the extensible SDK is a command-line tool called `devtool`. This tool provides a number of features that help you build, test and package software within the extensible SDK, and optionally integrate it into an image built by the OpenEmbedded build system.

> **Tip**
> The use of `devtool` is not limited to the extensible SDK. You can use `devtool` to help you easily develop any project whose build output must be part of an image built using the build system.

The `devtool` command line is organized similarly to Git in that it has a number of sub-commands for each function. You can run `devtool --help` to see all the commands.

Note

See the "devtool Quick Reference" in the Yocto Project Reference Manual for a devtool quick reference.

Three devtool subcommands exist that provide entry-points into development:

- ***devtool add***: Assists in adding new software to be built.
- ***devtool modify***: Sets up an environment to enable you to modify the source of an existing component.
- ***devtool upgrade***: Updates an existing recipe so that you can build it for an updated set of source files.

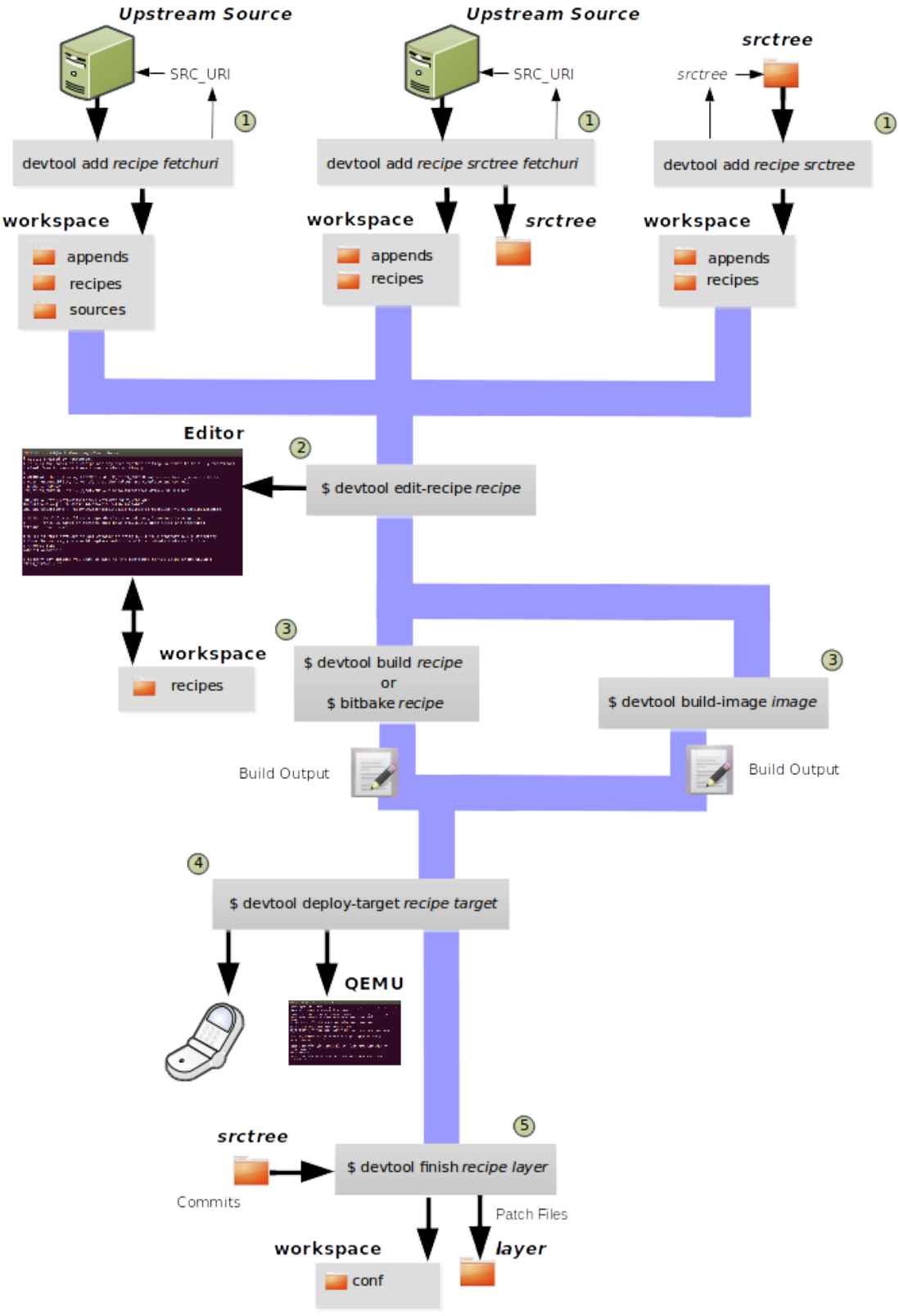
As with the build system, "recipes" represent software packages within devtool. When you use devtool add, a recipe is automatically created. When you use devtool modify, the specified existing recipe is used in order to determine where to get the source code and how to patch it. In both cases, an environment is set up so that when you build the recipe a source tree that is under your control is used in order to allow you to make changes to the source as desired. By default, new recipes and the source go into a "workspace" directory under the SDK.

The remainder of this section presents the devtool add, devtool modify, and devtool upgrade workflows.

2.4.1. Use devtool add to Add an Application¶

The devtool add command generates a new recipe based on existing source code. This command takes advantage of the workspace layer that many devtool commands use. The command is flexible enough to allow you to extract source code into both the workspace or a separate local Git repository and to use existing code that does not need to be extracted.

Depending on your particular scenario, the arguments and options you use with devtool add form different combinations. The following diagram shows common development flows you would use with the devtool add command:



1. ***Generating the New Recipe***: The top part of the flow shows three scenarios by which you could use `devtool add` to generate a recipe based on existing source code.

   In a shared development environment, it is typical for other developers to be responsible for various areas of source code. As a developer, you are probably interested in using that source code as part of your development within the Yocto Project. All you need is access to the code, a recipe, and a controlled area in which to do your work.

   Within the diagram, three possible scenarios feed into the `devtool add` workflow:

   - ***Left***: The left scenario in the figure represents a common situation where the source code does not exist locally and needs to be extracted. In this situation, the source code is extracted to the default workspace - you do not want the files in some specific location outside of the workspace. Thus, everything you need will be located in the workspace:

```
$ devtool add recipe fetchuri
```

With this command, devtool extracts the upstream source files into a local Git repository within the sources folder. The command then creates a recipe named *recipe* and a corresponding append file in the workspace. If you do not provide *recipe*, the command makes an attempt to determine the recipe name.

- ***Middle***: The middle scenario in the figure also represents a situation where the source code does not exist locally. In this case, the code is again upstream and needs to be extracted to some local area - this time outside of the default workspace.

> **Note**
> If required, devtool always creates a Git repository locally during the extraction.

Furthermore, the first positional argument *srctree* in this case identifies where the devtool add command will locate the extracted code outside of the workspace. You need to specify an empty directory:

```
$ devtool add recipe srctree fetchuri
```

In summary, the source code is pulled from *fetchuri* and extracted into the location defined by *srctree* as a local Git repository.

Within workspace, devtool creates a recipe named *recipe* along with an associated append file.

- ***Right***: The right scenario in the figure represents a situation where the *srctree* has been previously prepared outside of the devtool workspace.

  The following command provides a new recipe name and identifies the existing source tree location:

  ```
  $ devtool add recipe srctree
  ```

  The command examines the source code and creates a recipe named *recipe* for the code and places the recipe into the workspace.

  Because the extracted source code already exists, devtool does not try to relocate the source code into the workspace - only the new recipe is placed in the workspace.

  Aside from a recipe folder, the command also creates an associated append folder and places an initial *.bbappend file within.

2. ***Edit the Recipe***: You can use devtool edit-recipe to open up the editor as defined by the $EDITOR environment variable and modify the file:

   ```
   $ devtool edit-recipe recipe
   ```

   From within the editor, you can make modifications to the recipe that take affect when you build it later.

3. ***Build the Recipe or Rebuild the Image***: The next step you take depends on what you are going to do with the new code.

   If you need to eventually move the build output to the target hardware, use the following devtool command:

   ```
   $ devtool build recipe
   ```

   On the other hand, if you want an image to contain the recipe's packages from the workspace for immediate deployment onto a device (e.g. for testing purposes), you can use the devtool build-image command:

   ```
   $ devtool build-image image
   ```

4. ***Deploy the Build Output***: When you use the devtool build command to build out your recipe, you probably want to see if the resulting build output works as expected on the target hardware.

   > **Note**
   > This step assumes you have a previously built image that is already either running in QEMU or is running on actual hardware. Also, it is assumed that for deployment of the image to the target, SSH is installed in the image and, if the image is running on real hardware, you have network access to and from your development machine.

   You can deploy your build output to that target hardware by using the devtool deploy-target command:

   ```
   $ devtool deploy-target recipe target
   ```

   The *target* is a live target machine running as an SSH server.

   You can, of course, also deploy the image you build to actual hardware by using the devtool build-image command. However, devtool does not provide a specific command that allows you to deploy the image to actual

hardware.

5. ***Finish Your Work With the Recipe***: The `devtool finish` command creates any patches corresponding to commits in the local Git repository, moves the new recipe to a more permanent layer, and then resets the recipe so that the recipe is built normally rather than from the workspace.

   ```
   $ devtool finish recipe layer
   ```

   Note
   Any changes you want to turn into patches must be committed to the Git repository in the source tree.

   As mentioned, the `devtool finish` command moves the final recipe to its permanent layer.

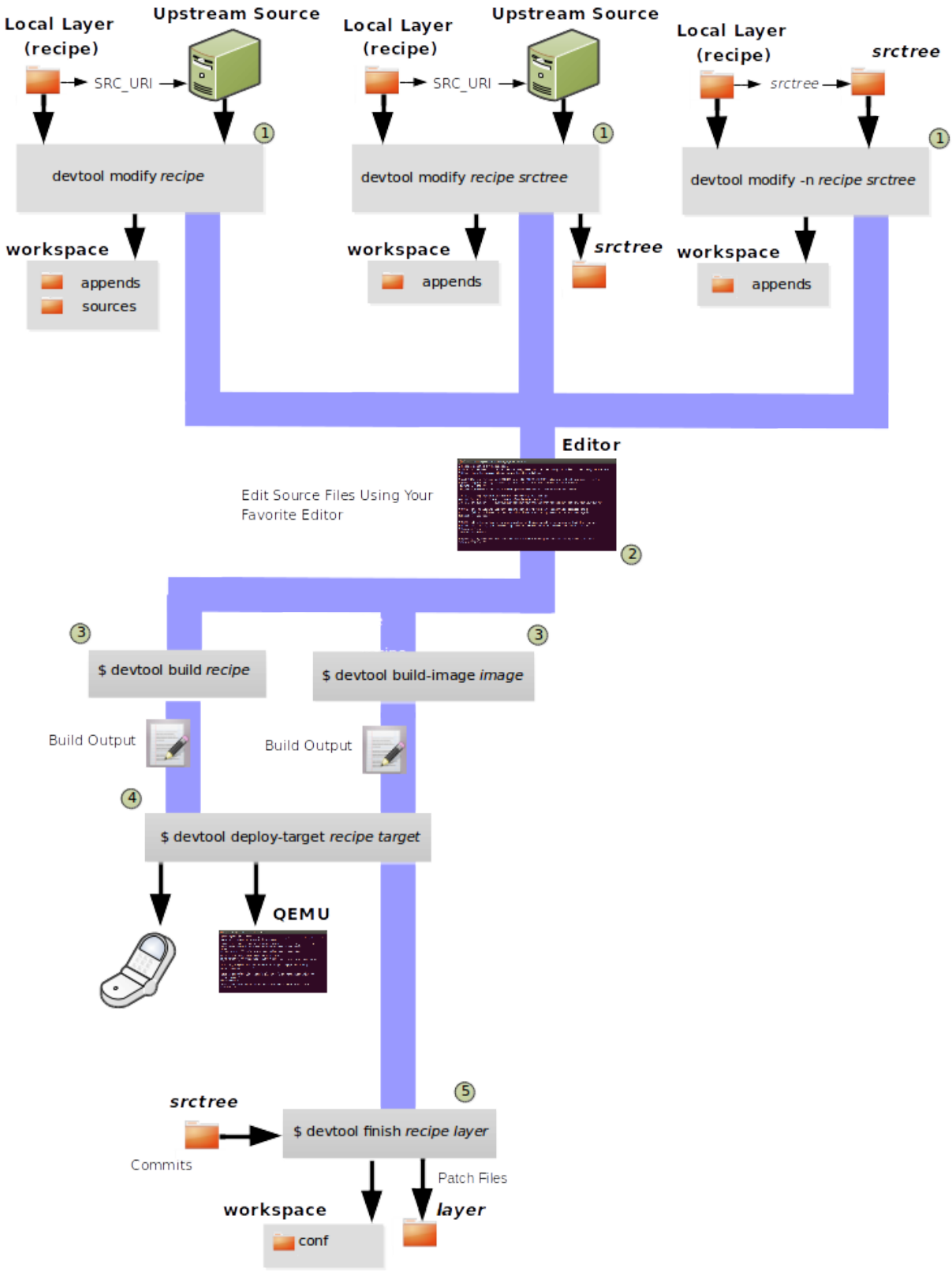
   As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than the workspace.

   Note
   You can use the `devtool reset` command to put things back should you decide you do not want to proceed with your work. If you do use this command, realize that the source tree is preserved.

2.4.2. Use `devtool modify` to Modify the Source of an Existing Component¶

The `devtool modify` command prepares the way to work on existing code that already has a local recipe in place that is used to build the software. The command is flexible enough to allow you to extract code from an upstream source, specify the existing recipe, and keep track of and gather any patch files from other developers that are associated with the code.

Depending on your particular scenario, the arguments and options you use with `devtool modify` form different combinations. The following diagram shows common development flows for the `devtool modify` command:



1. ***Preparing to Modify the Code***: The top part of the flow shows three scenarios by which you could use `devtool modify` to prepare to work on source files. Each scenario assumes the following:

   - The recipe exists locally in a layer external to the `devtool` workspace.
   - The source files exist either upstream in an un-extracted state or locally in a previously extracted state.

   The typical situation is where another developer has created a layer for use with the Yocto Project and their recipe already resides in that layer. Furthermore, their source code is readily available either upstream or locally.

   - ***Left***: The left scenario in the figure represents a common situation where the source code does not exist locally and it needs to be extracted from an upstream source. In this situation, the source is extracted into the default `devtool` workspace location. The recipe, in this scenario, is in its own layer outside the workspace (i.e. `meta-`*`Layername`*).

The following command identifies the recipe and, by default, extracts the source files:

```
$ devtool modify recipe
```

Once devtool locates the recipe, devtool uses the recipe's SRC_URI statements to locate the source code and any local patch files from other developers.

With this scenario, no *srctree* argument exists. Consequently, the default behavior of the devtool modify command is to extract the source files pointed to by the SRC_URI statements into a local Git structure. Furthermore, the location for the extracted source is the default area within the devtool workspace. The result is that the command sets up both the source code and an append file within the workspace while the recipe remains in its original location.

- ***Middle***: The middle scenario in the figure represents a situation where the source code also does not exist locally. In this case, the code is again upstream and needs to be extracted to some local area as a Git repository. The recipe, in this scenario, is again local and in its own layer outside the workspace.

  The following command tells devtool what recipe with which to work and, in this case, identifies a local area for the extracted source files that is outside of the default devtool workspace:

  ```
  $ devtool modify recipe srctree
  ```

  Note

  You cannot provide a URL for *srctree* using the devtool command.

  As with all extractions, the command uses the recipe's SRC_URI statements to locate the source files and any associated patch files. Once the files are located, the command by default extracts them into *srctree*.

  Within workspace, devtool creates an append file for the recipe. The recipe remains in its original location but the source files are extracted to the location you provide with *srctree*.

- ***Right***: The right scenario in the figure represents a situation where the source tree (*srctree*) already exists locally as a previously extracted Git structure outside of the devtool workspace. In this example, the recipe also exists elsewhere locally in its own layer.

  The following command tells devtool the recipe with which to work, uses the "-n" option to indicate source does not need to be extracted, and uses *srctree* to point to the previously extracted source files:

  ```
  $ devtool modify -n recipe srctree
  ```

  Once the command finishes, it creates only an append file for the recipe in the devtool workspace. The recipe and the source code remain in their original locations.

2. ***Edit the Source***: Once you have used the devtool modify command, you are free to make changes to the source files. You can use any editor you like to make and save your source code modifications.

3. ***Build the Recipe or Rebuild the Image***: The next step you take depends on what you are going to do with the new code.

   If you need to eventually move the build output to the target hardware, use the following devtool command:

   ```
   $ devtool build recipe
   ```

   On the other hand, if you want an image to contain the recipe's packages from the workspace for immediate deployment onto a device (e.g. for testing purposes), you can use the devtool build-image command:

   ```
   $ devtool build-image image
   ```

4. ***Deploy the Build Output***: When you use the devtool build command to build out your recipe, you probably want to see if the resulting build output works as expected on target hardware.

   Note

   This step assumes you have a previously built image that is already either running in QEMU or running on actual hardware. Also, it is assumed that for deployment of the image to the target, SSH is installed in the image and if the image is running on real hardware that you have network access to and from your development machine.

   You can deploy your build output to that target hardware by using the devtool deploy-target command:

   ```
   $ devtool deploy-target recipe target
   ```

   The *target* is a live target machine running as an SSH server.

   You can, of course, use other methods to deploy the image you built using the devtool build-image command to actual hardware. devtool does not provide a specific command to deploy the image to actual hardware.

5. ***Finish Your Work With the Recipe***: The `devtool finish` command creates any patches corresponding to commits in the local Git repository, updates the recipe to point to them (or creates a `.bbappend` file to do so, depending on the specified destination layer), and then resets the recipe so that the recipe is built normally rather than from the workspace.

   ```
   $ devtool finish recipe layer
   ```

   Note

   Any changes you want to turn into patches must be staged and committed within the local Git repository before you use the `devtool finish` command.

   Because there is no need to move the recipe, `devtool finish` either updates the original recipe in the original layer or the command creates a `.bbappend` file in a different layer as provided by *layer*.

   As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than from the workspace.

   Note

   You can use the `devtool reset` command to put things back should you decide you do not want to proceed with your work. If you do use this command, realize that the source tree is preserved.

2.4.3. Use `devtool upgrade` to Create a Version of the Recipe that Supports a Newer Version of the Software¶

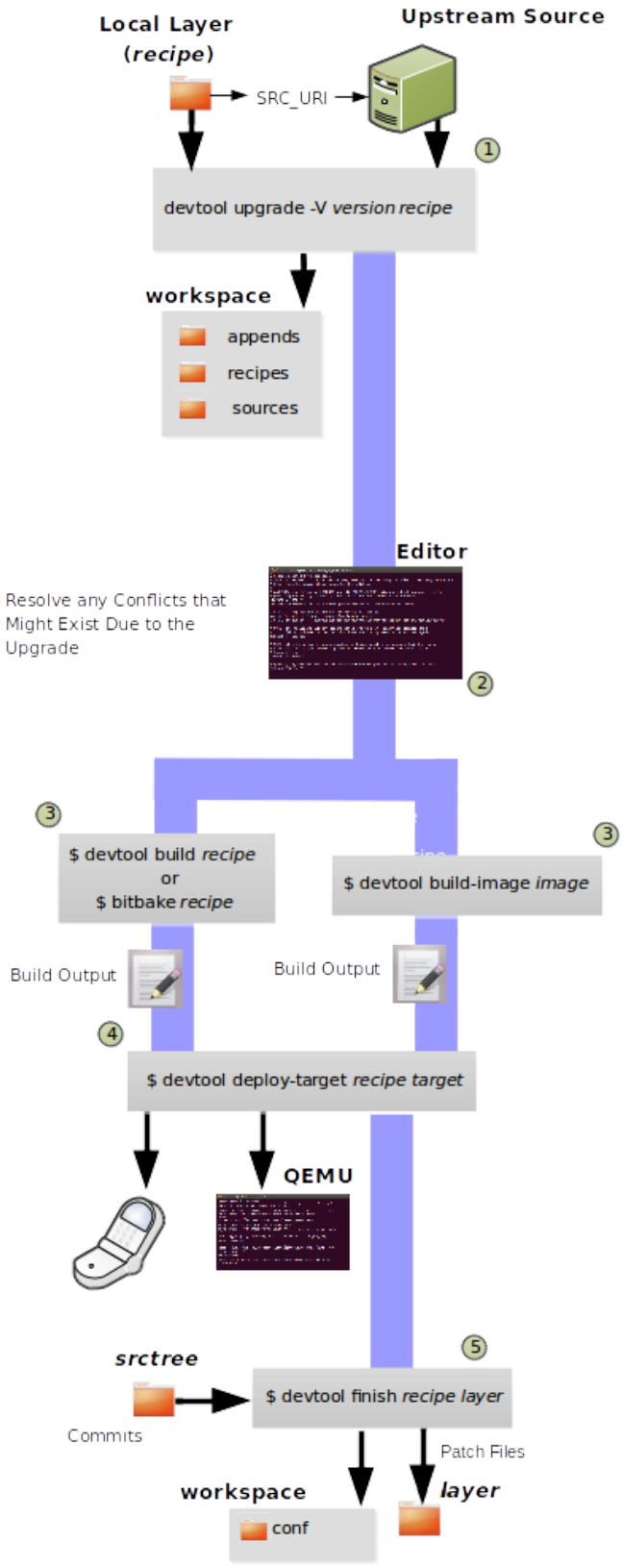
The `devtool upgrade` command upgrades an existing recipe to that of a more up-to-date version found upstream. Throughout the life of software, recipes continually undergo version upgrades by their upstream publishers. You can use the `devtool upgrade` workflow to make sure your recipes you are using for builds are up-to-date with their upstream counterparts.

Note

Several methods exist by which you can upgrade recipes - `devtool upgrade` happens to be one. You can read about all the methods by which you can upgrade recipes in the "Upgrading Recipes" section of the Yocto Project Development Tasks Manual.

The `devtool upgrade` command is flexible enough to allow you to specify source code revision and versioning schemes, extract code into or out of the `devtool` workspace, and work with any source file forms that the fetchers support.

The following diagram shows the common development flow used with the `devtool upgrade` command:



1. ***Initiate the Upgrade***: The top part of the flow shows the typical scenario by which you use the `devtool upgrade` command. The following conditions exist:

   - The recipe exists in a local layer external to the `devtool` workspace.
   - The source files for the new release exist in the same location pointed to by `SRC_URI` in the recipe (e.g. a tarball with the new version number in the name, or as a different revision in the upstream Git repository).

   A common situation is where third-party software has undergone a revision so that it has been upgraded. The recipe you have access to is likely in your own layer. Thus, you need to upgrade the recipe to use the newer version of the software:

   ```
   $ devtool upgrade -V version recipe
   ```

By default, the `devtool upgrade` command extracts source code into the `sources` directory in the workspace. If you want the code extracted to any other location, you need to provide the *srctree* positional argument with the command as follows:

```
$ devtool upgrade -V version recipe srctree
```

> **Note**
> In this example, the "-V" option specifies the new version. If you don't use "-V", the command upgrades the recipe to the latest version.

If the source files pointed to by the `SRC_URI` statement in the recipe are in a Git repository, you must provide the "-S" option and specify a revision for the software.

Once `devtool` locates the recipe, it uses the `SRC_URI` variable to locate the source code and any local patch files from other developers. The result is that the command sets up the source code, the new version of the recipe, and an append file all within the workspace.

2. ***Resolve any Conflicts created by the Upgrade***: Conflicts could exist due to the software being upgraded to a new version. Conflicts occur if your recipe specifies some patch files in `SRC_URI` that conflict with changes made in the new version of the software. For such cases, you need to resolve the conflicts by editing the source and following the normal `git rebase` conflict resolution process.

   Before moving onto the next step, be sure to resolve any such conflicts created through use of a newer or different version of the software.

3. ***Build the Recipe or Rebuild the Image***: The next step you take depends on what you are going to do with the new code.

   If you need to eventually move the build output to the target hardware, use the following `devtool` command:

   ```
   $ devtool build recipe
   ```

   On the other hand, if you want an image to contain the recipe's packages from the workspace for immediate deployment onto a device (e.g. for testing purposes), you can use the `devtool build-image` command:

   ```
   $ devtool build-image image
   ```

4. ***Deploy the Build Output***: When you use the `devtool build` command or `bitbake` to build your recipe, you probably want to see if the resulting build output works as expected on target hardware.

   > **Note**
   > This step assumes you have a previously built image that is already either running in QEMU or running on actual hardware. Also, it is assumed that for deployment of the image to the target, SSH is installed in the image and if the image is running on real hardware that you have network access to and from your development machine.

   You can deploy your build output to that target hardware by using the `devtool deploy-target` command:

   ```
   $ devtool deploy-target recipe target
   ```

   The *target* is a live target machine running as an SSH server.

   You can, of course, also deploy the image you build using the `devtool build-image` command to actual hardware. However, `devtool` does not provide a specific command that allows you to do this.

5. ***Finish Your Work With the Recipe***: The `devtool finish` command creates any patches corresponding to commits in the local Git repository, moves the new recipe to a more permanent layer, and then resets the recipe so that the recipe is built normally rather than from the workspace. If you specify a destination layer that is the same as the original source, then the old version of the recipe and associated files will be removed prior to adding the new version.

   ```
   $ devtool finish recipe layer
   ```

   > **Note**
   > Any changes you want to turn into patches must be committed to the Git repository in the source tree.

   As a final process of the `devtool finish` command, the state of the standard layers and the upstream source is restored so that you can build the recipe from those areas rather than the workspace.

> **Note**

You can use the `devtool reset` command to put things back should you decide you do not want to proceed with your work. If you do use this command, realize that the source tree is preserved.

2.5. A Closer Look at `devtool add`¶

The `devtool add` command automatically creates a recipe based on the source tree you provide with the command. Currently, the command has support for the following:

- Autotools (`autoconf` and `automake`)
- CMake
- Scons
- `qmake`
- Plain `Makefile`
- Out-of-tree kernel module
- Binary package (i.e. "-b" option)
- Node.js module
- Python modules that use `setuptools` or `distutils`

Apart from binary packages, the determination of how a source tree should be treated is automatic based on the files present within that source tree. For example, if a `CMakeLists.txt` file is found, then the source tree is assumed to be using CMake and is treated accordingly.

Note

In most cases, you need to edit the automatically generated recipe in order to make it build properly. Typically, you would go through several edit and build cycles until the recipe successfully builds. Once the recipe builds, you could use possible further iterations to test the recipe on the target device.

The remainder of this section covers specifics regarding how parts of the recipe are generated.

2.5.1. Name and Version¶

If you do not specify a name and version on the command line, `devtool add` uses various metadata within the source tree in an attempt to determine the name and version of the software being built. Based on what the tool determines, `devtool` sets the name of the created recipe file accordingly.

If `devtool` cannot determine the name and version, the command prints an error. For such cases, you must re-run the command and provide the name and version, just the name, or just the version as part of the command line.

Sometimes the name or version determined from the source tree might be incorrect. For such a case, you must reset the recipe:

```
$ devtool reset -n recipename
```

After running the `devtool reset` command, you need to run `devtool add` again and provide the name or the version.

2.5.2. Dependency Detection and Mapping¶

The `devtool add` command attempts to detect build-time dependencies and map them to other recipes in the system. During this mapping, the command fills in the names of those recipes as part of the `DEPENDS` variable within the recipe. If a dependency cannot be mapped, `devtool` places a comment in the recipe indicating such. The inability to map a dependency can result from naming not being recognized or because the dependency simply is not available. For cases where the dependency is not available, you must use the `devtool add` command to add an additional recipe that satisfies the dependency. Once you add that recipe, you need to update the `DEPENDS` variable in the original recipe to include the new recipe.

If you need to add runtime dependencies, you can do so by adding the following to your recipe:

```
RDEPENDS_${PN} += "dependency1 dependency2 ..."
```

Note

The `devtool add` command often cannot distinguish between mandatory and optional dependencies. Consequently, some of the detected dependencies might in fact be optional. When in doubt, consult the documentation or the configure script for the software the recipe is building for further details. In some cases, you might find you can substitute the dependency with an option that disables the associated functionality passed to the configure script.

2.5.3. License Detection¶

The `devtool add` command attempts to determine if the software you are adding is able to be distributed under a common, open-source license. If so, the command sets the `LICENSE` value accordingly. You should double-check the value added by the command against the documentation or source files for the software you are building and, if necessary, update that `LICENSE` value.

The `devtool add` command also sets the `LIC_FILES_CHKSUM` value to point to all files that appear to be license-related. Realize that license statements often appear in comments at the top of source files or within the documentation. In such cases, the command does not recognize those license statements. Consequently, you might need to amend the `LIC_FILES_CHKSUM` variable to point to one or more of those comments if present. Setting `LIC_FILES_CHKSUM` is particularly important for third-party software. The mechanism attempts to ensure correct licensing should you upgrade the recipe to a newer upstream version in future. Any change in licensing is detected and you receive an error prompting you to check the license text again.

If the `devtool add` command cannot determine licensing information, `devtool` sets the `LICENSE` value to "CLOSED" and leaves the `LIC_FILES_CHKSUM` value unset. This behavior allows you to continue with development even though the settings are unlikely to be correct in all cases. You should check the documentation or source files for the software you are building to determine the actual license.

2.5.4. Adding Makefile-Only Software¶

The use of Make by itself is very common in both proprietary and open-source software. Unfortunately, Makefiles are often not written with cross-compilation in mind. Thus, `devtool add` often cannot do very much to ensure that these Makefiles build correctly. It is very common, for example, to explicitly call `gcc` instead of using the `CC` variable. Usually, in a cross-compilation environment, `gcc` is the compiler for the build host and the cross-compiler is named something similar to `arm-poky-linux-gnueabi-gcc` and might require arguments (e.g. to point to the associated sysroot for the target machine).

When writing a recipe for Makefile-only software, keep the following in mind:

- You probably need to patch the Makefile to use variables instead of hardcoding tools within the toolchain such as `gcc` and `g++`.
- The environment in which Make runs is set up with various standard variables for compilation (e.g. `CC`, `CXX`, and so forth) in a similar manner to the environment set up by the SDK's environment setup script. One easy way to see these variables is to run the `devtool build` command on the recipe and then look in `oe-logs/run.do_compile`. Towards the top of this file, a list of environment variables exists that are being set. You can take advantage of these variables within the Makefile.
- If the Makefile sets a default for a variable using "=", that default overrides the value set in the environment, which is usually not desirable. For this case, you can either patch the Makefile so it sets the default using the "?=" operator, or you can alternatively force the value on the `make` command line. To force the value on the command line, add the variable setting to `EXTRA_OEMAKE` or `PACKAGECONFIG_CONFARGS` within the recipe. Here is an example using `EXTRA_OEMAKE`:

  ```
  EXTRA_OEMAKE += "'CC=${CC}' 'CXX=${CXX}'"
  ```

  In the above example, single quotes are used around the variable settings as the values are likely to contain spaces because required default options are passed to the compiler.
- Hardcoding paths inside Makefiles is often problematic in a cross-compilation environment. This is particularly true because those hardcoded paths often point to locations on the build host and thus will either be read-only or will introduce contamination into the cross-compilation because they are specific to the build host rather than the target. Patching the Makefile to use prefix variables or other path variables is usually the way to handle this situation.
- Sometimes a Makefile runs target-specific commands such as `ldconfig`. For such cases, you might be able to apply patches that remove these commands from the Makefile.

2.5.5. Adding Native Tools¶

Often, you need to build additional tools that run on the build host as opposed to the target. You should indicate this requirement by using one of the following methods when you run `devtool add`:

- Specify the name of the recipe such that it ends with "-native". Specifying the name like this produces a recipe that only builds for the build host.
- Specify the "--also-native" option with the `devtool add` command. Specifying this option creates a recipe file that still builds for the target but also creates a variant with a "-native" suffix that builds for the build host.

> **Note**
> If you need to add a tool that is shipped as part of a source tree that builds code for the target, you can typically accomplish this by building the native and target parts separately rather than within the same compilation process. Realize though that with the "--also-native" option, you can add the tool using just one recipe file.

2.5.6. Adding Node.js Modules¶

You can use the `devtool add` command two different ways to add Node.js modules: 1) Through `npm` and, 2) from a repository or local source.

Use the following form to add Node.js modules through `npm`:

```
$ devtool add "npm://registry.npmjs.org;name=forever;version=0.15.1"
```

The name and version parameters are mandatory. Lockdown and shrinkwrap files are generated and pointed to by the recipe in order to freeze the version that is fetched for the dependencies according to the first time. This also saves checksums that are verified on future fetches. Together, these behaviors ensure the reproducibility and integrity of the build.

Notes

- You must use quotes around the URL. The `devtool add` does not require the quotes, but the shell considers ";" as a splitter between multiple commands. Thus, without the quotes, `devtool add` does not receive the other parts, which results in several "command not found" errors.
- In order to support adding Node.js modules, a `nodejs` recipe must be part of your SDK.

As mentioned earlier, you can also add Node.js modules directly from a repository or local source tree. To add modules this way, use `devtool add` in the following form:

```
$ devtool add https://github.com/diversario/node-ssdp
```

In this example, `devtool` fetches the specified Git repository, detects the code as Node.js code, fetches dependencies using `npm`, and sets `SRC_URI` accordingly.

2.6. Working With Recipes¶

When building a recipe using the `devtool build` command, the typical build progresses as follows:

1. Fetch the source
2. Unpack the source
3. Configure the source
4. Compile the source
5. Install the build output
6. Package the installed output

For recipes in the workspace, fetching and unpacking is disabled as the source tree has already been prepared and is persistent. Each of these build steps is defined as a function (task), usually with a "do_" prefix (e.g. `do_fetch`, `do_unpack`, and so forth). These functions are typically shell scripts but can instead be written in Python.

If you look at the contents of a recipe, you will see that the recipe does not include complete instructions for building the software. Instead, common functionality is encapsulated in classes inherited with the `inherit` directive. This technique leaves the recipe to describe just the things that are specific to the software being built. A `base` class exists that is implicitly inherited by all recipes and provides the functionality that most recipes typically need.

The remainder of this section presents information useful when working with recipes.

2.6.1. Finding Logs and Work Files¶

After the first run of the `devtool build` command, recipes that were previously created using the `devtool add` command or whose sources were modified using the `devtool modify` command contain symbolic links created within the source tree:

- `oe-logs`: This link points to the directory in which log files and run scripts for each build step are created.
- `oe-workdir`: This link points to the temporary work area for the recipe. The following locations under `oe-workdir` are particularly useful:
  - `image/`: Contains all of the files installed during the `do_install` stage. Within a recipe, this directory is referred to by the expression `${D}`.
  - `sysroot-destdir/`: Contains a subset of files installed within `do_install` that have been put into the shared sysroot. For more information, see the "Sharing Files Between Recipes" section.
  - `packages-split/`: Contains subdirectories for each package produced by the recipe. For more information, see the "Packaging" section.

You can use these links to get more information on what is happening at each build step.

2.6.2. Setting Configure Arguments¶

If the software your recipe is building uses GNU autoconf, then a fixed set of arguments is passed to it to enable cross-compilation plus any extras specified by EXTRA_OECONF or PACKAGECONFIG_CONFARGS set within the recipe. If you wish to pass additional options, add them to EXTRA_OECONF or PACKAGECONFIG_CONFARGS. Other supported build tools have similar variables (e.g. EXTRA_OECMAKE for CMake, EXTRA_OESCONS for Scons, and so forth). If you need to pass anything on the make command line, you can use EXTRA_OEMAKE or the PACKAGECONFIG_CONFARGS variables to do so.

You can use the devtool configure-help command to help you set the arguments listed in the previous paragraph. The command determines the exact options being passed, and shows them to you along with any custom arguments specified through EXTRA_OECONF or PACKAGECONFIG_CONFARGS. If applicable, the command also shows you the output of the configure script's "--help" option as a reference.

2.6.3. Sharing Files Between Recipes¶

Recipes often need to use files provided by other recipes on the build host. For example, an application linking to a common library needs access to the library itself and its associated headers. The way this access is accomplished within the extensible SDK is through the sysroot. One sysroot exists per "machine" for which the SDK is being built. In practical terms, this means a sysroot exists for the target machine, and a sysroot exists for the build host.

Recipes should never write files directly into the sysroot. Instead, files should be installed into standard locations during the do_install task within the ${D} directory. A subset of these files automatically goes into the sysroot. The reason for this limitation is that almost all files that go into the sysroot are cataloged in manifests in order to ensure they can be removed later when a recipe is modified or removed. Thus, the sysroot is able to remain free from stale files.

2.6.4. Packaging¶

Packaging is not always particularly relevant within the extensible SDK. However, if you examine how build output gets into the final image on the target device, it is important to understand packaging because the contents of the image are expressed in terms of packages and not recipes.

During the do_package task, files installed during the do_install task are split into one main package, which is almost always named the same as the recipe, and into several other packages. This separation exists because not all of those installed files are useful in every image. For example, you probably do not need any of the documentation installed in a production image. Consequently, for each recipe the documentation files are separated into a -doc package. Recipes that package software containing optional modules or plugins might undergo additional package splitting as well.

After building a recipe, you can see where files have gone by looking in the oe-workdir/packages-split directory, which contains a subdirectory for each package. Apart from some advanced cases, the PACKAGES and FILES variables controls splitting. The PACKAGES variable lists all of the packages to be produced, while the FILES variable specifies which files to include in each package by using an override to specify the package. For example, FILES_${PN} specifies the files to go into the main package (i.e. the main package has the same name as the recipe and ${PN} evaluates to the recipe name). The order of the PACKAGES value is significant. For each installed file, the first package whose FILES value matches the file is the package into which the file goes. Defaults exist for both the PACKAGES and FILES variables. Consequently, you might find you do not even need to set these variables in your recipe unless the software the recipe is building installs files into non-standard locations.

2.7. Restoring the Target Device to its Original State¶

If you use the devtool deploy-target command to write a recipe's build output to the target, and you are working on an existing component of the system, then you might find yourself in a situation where you need to restore the original files that existed prior to running the devtool deploy-target command. Because the devtool deploy-target command backs up any files it overwrites, you can use the devtool undeploy-target command to restore those files and remove any other files the recipe deployed. Consider the following example:

```
$ devtool undeploy-target lighttpd root@192.168.7.2
```

If you have deployed multiple applications, you can remove them all using the "-a" option thus restoring the target device to its original state:

```
$ devtool undeploy-target -a root@192.168.7.2
```

Information about files deployed to the target as well as any backed up files are stored on the target itself. This storage, of course, requires some additional space on the target machine.

> **Note**
> The devtool deploy-target and devtool undeploy-target commands do not currently interact with any package management system on the target device (e.g. RPM or OPKG). Consequently, you should not intermingle devtool deploy-target and package manager operations on the target device. Doing so could result in a conflicting set of files.

2.8. Installing Additional Items Into the Extensible SDK¶

Out of the box the extensible SDK typically only comes with a small number of tools and libraries. A minimal SDK starts mostly empty and is populated on-demand. Sometimes you must explicitly install extra items into the SDK. If you need these extra

items, you can first search for the items using the `devtool search` command. For example, suppose you need to link to libGL but you are not sure which recipe provides libGL. You can use the following command to find out:

```
$ devtool search libGL
mesa                A free implementation of the OpenGL API
```

Once you know the recipe (i.e. `mesa` in this example), you can install it:

```
$ devtool sdk-install mesa
```

By default, the `devtool sdk-install` command assumes the item is available in pre-built form from your SDK provider. If the item is not available and it is acceptable to build the item from source, you can add the "-s" option as follows:

```
$ devtool sdk-install -s mesa
```

It is important to remember that building the item from source takes significantly longer than installing the pre-built artifact. Also, if no recipe exists for the item you want to add to the SDK, you must instead add the item using the `devtool add` command.

2.9. Applying Updates to an Installed Extensible SDK¶

If you are working with an installed extensible SDK that gets occasionally updated (e.g. a third-party SDK), then you will need to manually "pull down" the updates into the installed SDK.

To update your installed SDK, use `devtool` as follows:

```
$ devtool sdk-update
```

The previous command assumes your SDK provider has set the default update URL for you through the `SDK_UPDATE_URL` variable as described in the "Providing Updates to the Extensible SDK After Installation" section. If the SDK provider has not set that default URL, you need to specify it yourself in the command as follows:

```
$ devtool sdk-update path_to_update_directory
```

> **Note**
> The URL needs to point specifically to a published SDK and not to an SDK installer that you would download and install.

2.10. Creating a Derivative SDK With Additional Components¶

You might need to produce an SDK that contains your own custom libraries. A good example would be if you were a vendor with customers that use your SDK to build their own platform-specific software and those customers need an SDK that has custom libraries. In such a case, you can produce a derivative SDK based on the currently installed SDK fairly easily by following these steps:

1. If necessary, install an extensible SDK that you want to use as a base for your derivative SDK.
2. Source the environment script for the SDK.
3. Add the extra libraries or other components you want by using the `devtool add` command.
4. Run the `devtool build-sdk` command.

The previous steps take the recipes added to the workspace and construct a new SDK installer that contains those recipes and the resulting binary artifacts. The recipes go into their own separate layer in the constructed derivative SDK, which leaves the workspace clean and ready for users to add their own recipes.

Chapter 3. Using the Standard SDK¶

Table of Contents

This chapter describes the standard SDK and how to install it. Information includes unique installation and setup aspects for the standard SDK.

> **Note**
> For a side-by-side comparison of main features supported for a standard SDK as compared to an extensible SDK, see the "Introduction" section.

You can use a standard SDK to work on Makefile, Autotools, and Eclipse™-based projects. See the "Using the SDK Toolchain Directly" chapter for more information.

3.1. Why use the Standard SDK and What is in It?¶

The Standard SDK provides a cross-development toolchain and libraries tailored to the contents of a specific image. You would use the Standard SDK if you want a more traditional toolchain experience as compared to the extensible SDK, which provides an internal build system and the `devtool` functionality.

The installed Standard SDK consists of several files and directories. Basically, it contains an SDK environment setup script, some configuration files, and host and target root filesystems to support usage. You can see the directory structure in the "Installed Standard SDK Directory Structure" section.

3.2. Installing the SDK¶

The first thing you need to do is install the SDK on your Build Host by running the `*.sh` installation script.

You can download a tarball installer, which includes the pre-built toolchain, the `runqemu` script, and support files from the appropriate toolchain directory within the Index of Releases. Toolchains are available for several 32-bit and 64-bit architectures with the `x86_64` directories, respectively. The toolchains the Yocto Project provides are based off the `core-image-sato` and `core-image-minimal` images and contain libraries appropriate for developing against that image.

The names of the tarball installer scripts are such that a string representing the host system appears first in the filename and then is immediately followed by a string representing the target architecture.

```
poky-glibc-host_system-image_type-arch-toolchain-release_version.sh

Where:
    host_system is a string representing your development system:

               i686 or x86_64.

    image_type is the image for which the SDK was built:

               core-image-minimal or core-image-sato.

    arch is a string representing the tuned target architecture:

               aarch64, armv5e, core2-64, i586, mips32r2, mips64, ppc7400, or cortexa8hf-neon.

    release_version is a string representing the release number of the Yocto Project:

               2.5, 2.5+snapshot
```

For example, the following SDK installer is for a 64-bit development host system and a i586-tuned target architecture based off the SDK for `core-image-sato` and using the current 2.5 snapshot:

```
poky-glibc-x86_64-core-image-sato-i586-toolchain-2.5.sh
```

> **Note**
> As an alternative to downloading an SDK, you can build the SDK installer. For information on building the installer, see the "Building an SDK Installer" section. Another helpful resource for building an installer is the Cookbook guide to Making an Eclipse Debug Capable Image wiki page. This wiki page focuses on development when using the Eclipse IDE.

The SDK and toolchains are self-contained and by default are installed into the `poky_sdk` folder in your home directory. You can choose to install the extensible SDK in any location when you run the installer. However, because files need to be written under that directory during the normal course of operation, the location you choose for installation must be writable for whichever users need to use the SDK.

The following command shows how to run the installer given a toolchain tarball for a 64-bit x86 development host system and a 64-bit x86 target architecture. The example assumes the SDK installer is located in `~/Downloads/` and has execution rights.

> **Note**
> If you do not have write permissions for the directory into which you are installing the SDK, the installer notifies you and exits. For that case, set up the proper permissions in the directory and run the installer again.

```
$ ./Downloads/poky-glibc-x86_64-core-image-sato-i586-toolchain-2.5.sh
Poky (Yocto Project Reference Distro) SDK installer version 2.5
===========================================================
Enter target directory for SDK (default: /opt/poky/2.5):
You are about to install the SDK to "/opt/poky/2.5". Proceed[Y/n]? Y
```

```
Extracting SDK..................................... ..........................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
 $ . /opt/poky/2.5/environment-setup-i586-poky-linux
```

Again, reference the "Installed Standard SDK Directory Structure" section for more details on the resulting directory structure of the installed SDK.

3.3. Running the SDK Environment Setup Script¶

Once you have the SDK installed, you must run the SDK environment setup script before you can actually use the SDK. This setup script resides in the directory you chose when you installed the SDK, which is either the default `/opt/poky/2.5` directory or the directory you chose during installation.

Before running the script, be sure it is the one that matches the architecture for which you are developing. Environment setup scripts begin with the string "`environment-setup`" and include as part of their name the tuned target architecture. As an example, the following commands set the working directory to where the SDK was installed and then source the environment setup script. In this example, the setup script is for an IA-based target machine using i586 tuning:

```
$ source /opt/poky/2.5/environment-setup-i586-poky-linux
```

When you run the setup script, the same environment variables are defined as are when you run the setup script for an extensible SDK. See the "Running the Extensible SDK Environment Setup Script" section for more information.

Chapter 4. Using the SDK Toolchain Directly¶

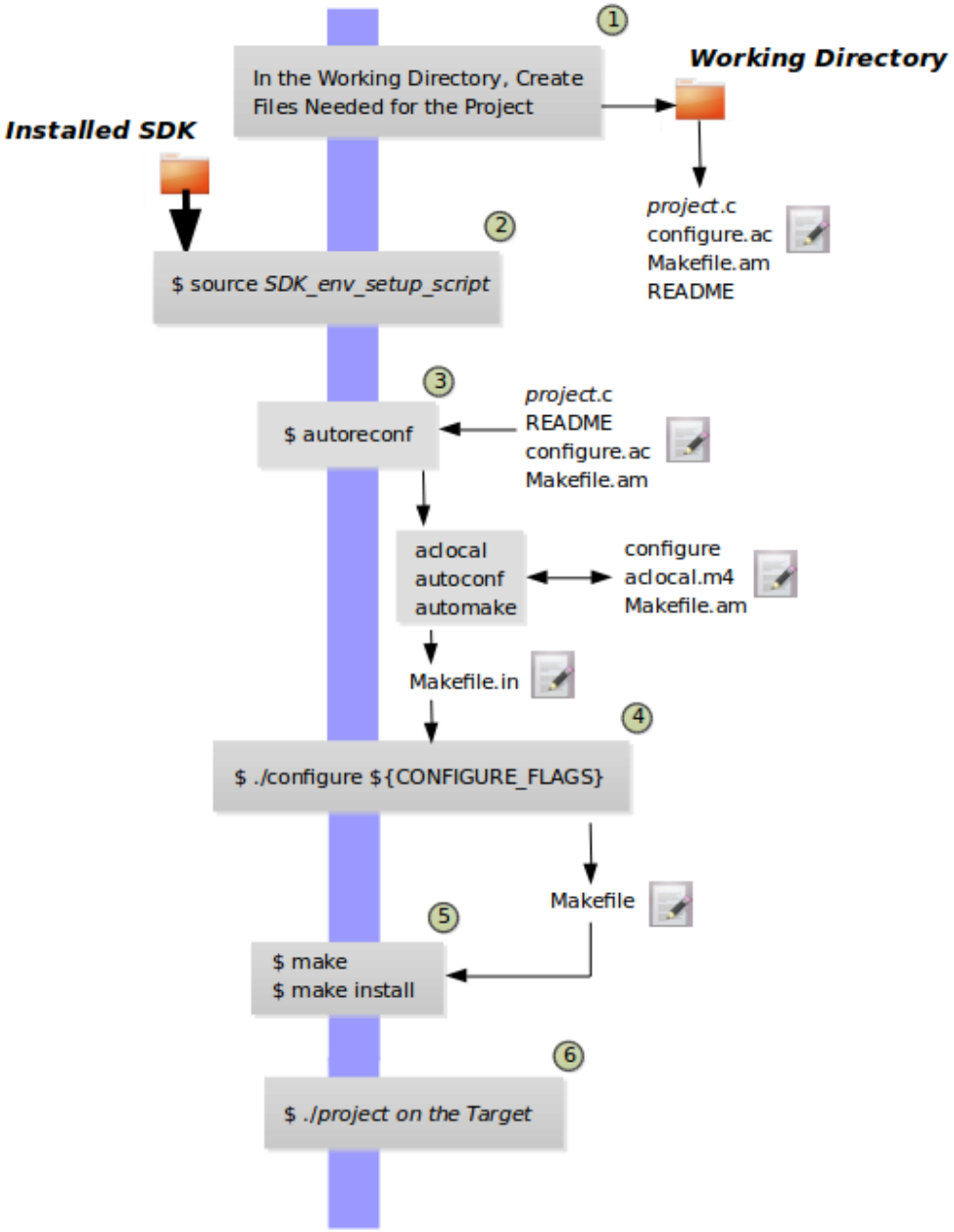
Table of Contents

You can use the SDK toolchain directly with Makefile, Autotools, and Eclipse™-based projects. This chapter covers the first two, while the "Developing Applications Using Eclipse™" Chapter covers the latter.

4.1. Autotools-Based Projects¶

Once you have a suitable cross-development toolchain installed, it is very easy to develop a project using the GNU Autotools-based workflow, which is outside of the OpenEmbedded build system.

The following figure presents a simple Autotools workflow.



Follow these steps to create a simple Autotools-based "Hello World" project:

Note

For more information on the GNU Autotools workflow, see the same example on the GNOME Developer site.

1. ***Create a Working Directory and Populate It:*** Create a clean directory for your project and then make that directory your working location.

```
$ mkdir $HOME/helloworld
$ cd $HOME/helloworld
```

After setting up the directory, populate it with files needed for the flow. You need a project source file, a file to help with configuration, and a file to help create the Makefile, and a README file: `hello.c`, `configure.ac`, `Makefile.am`, and `README`, respectively.

Use the following command to create an empty README file, which is required by GNU Coding Standards:

```
$ touch README
```

Create the remaining three files as follows:

- ***`hello.c`:***

```
#include <stdio.h>

main()
   {
      printf("Hello World!\n");
```

```
}
```

- ***configure.ac:***

```
AC_INIT(hello,0.1)
AM_INIT_AUTOMAKE([foreign])
AC_PROG_CC
AC_CONFIG_FILES(Makefile)
AC_OUTPUT
```

- ***Makefile.am:***

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

2. ***Source the Cross-Toolchain Environment Setup File:*** As described earlier in the manual, installing the cross-toolchain creates a cross-toolchain environment setup script in the directory that the SDK was installed. Before you can use the tools to develop your project, you must source this setup script. The script begins with the string "environment-setup" and contains the machine architecture, which is followed by the string "poky-linux". For this example, the command sources a script from the default SDK installation directory that uses the 32-bit Intel x86 Architecture and the Sumo Yocto Project release:

```
$ source /opt/poky/2.5/environment-setup-i586-poky-linux
```

3. ***Create the `configure` Script:*** Use the `autoreconf` command to generate the `configure` script.

```
$ autoreconf
```

The `autoreconf` tool takes care of running the other Autotools such as `aclocal`, `autoconf`, and `automake`.

> **Note**
> If you get errors from `configure.ac`, which `autoreconf` runs, that indicate missing files, you can use the "-i" option, which ensures missing auxiliary files are copied to the build host.

4. ***Cross-Compile the Project:*** This command compiles the project using the cross-compiler. The `CONFIGURE_FLAGS` environment variable provides the minimal arguments for GNU configure:

```
$ ./configure ${CONFIGURE_FLAGS}
```

For an Autotools-based project, you can use the cross-toolchain by just passing the appropriate host option to `configure.sh`. The host option you use is derived from the name of the environment setup script found in the directory in which you installed the cross-toolchain. For example, the host option for an ARM-based target that uses the GNU EABI is `armv5te-poky-linux-gnueabi`. You will notice that the name of the script is `environment-setup-armv5te-poky-linux-gnueabi`. Thus, the following command works to update your project and rebuild it using the appropriate cross-toolchain tools:

```
$ ./configure --host=armv5te-poky-linux-gnueabi --with-libtool-sysroot=sysroot_dir
```

5. ***Make and Install the Project:*** These two commands generate and install the project into the destination directory:

```
$ make
$ make install DESTDIR=./tmp
```

> **Note**
> To learn about environment variables established when you run the cross-toolchain environment setup script and how they are used or overridden when the Makefile, see the "Makefile-Based Projects" section.

This next command is a simple way to verify the installation of your project. Running the command prints the architecture on which the binary file can run. This architecture should be the same architecture that the installed cross-toolchain supports.

```
$ file ./tmp/usr/local/bin/hello
```

6. ***Execute Your Project:*** To execute the project, you would need to run it on your target hardware. If your target hardware happens to be your build host, you could run the project as follows:

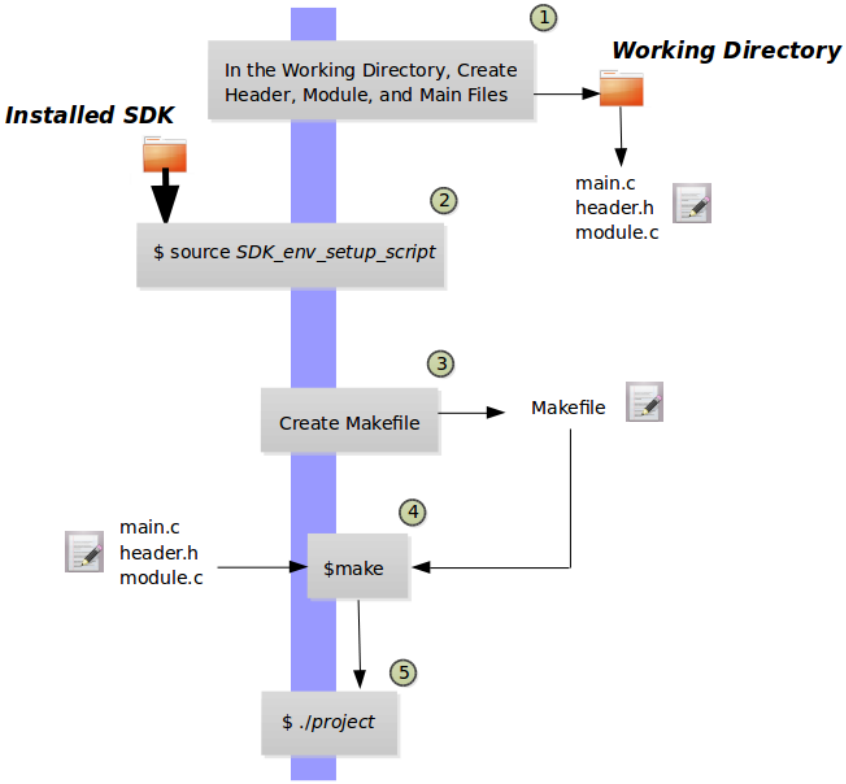
```
$ ./tmp/usr/local/bin/hello
```

As expected, the project displays the "Hello World!" message.

4.2. Makefile-Based Projects¶

Simple Makefile-based projects use and interact with the cross-toolchain environment variables established when you run the cross-toolchain environment setup script. The environment variables are subject to general `make` rules.

This section presents a simple Makefile development flow and provides an example that lets you see how you can use cross-toolchain environment variables and Makefile variables during development.



The main point of this section is to explain the following three cases regarding variable behavior:

- ***Case 1 - No Variables Set in the `Makefile` Map to Equivalent Environment Variables Set in the SDK Setup Script:*** Because matching variables are not specifically set in the `Makefile`, the variables retain their values based on the environment setup script.
- ***Case 2 - Variables Are Set in the Makefile that Map to Equivalent Environment Variables from the SDK Setup Script:*** Specifically setting matching variables in the `Makefile` during the build results in the environment settings of the variables being overwritten. In this case, the variables you set in the `Makefile` are used.
- ***Case 3 - Variables Are Set Using the Command Line that Map to Equivalent Environment Variables from the SDK Setup Script:*** Executing the `Makefile` from the command line results in the environment variables being overwritten. In this case, the command-line content is used.

Note

Regardless of how you set your variables, if you use the "-e" option with `make`, the variables from the SDK setup script take precedence:

```
$ make -e target
```

The remainder of this section presents a simple Makefile example that demonstrates these variable behaviors.

In a new shell environment variables are not established for the SDK until you run the setup script. For example, the following commands show a null value for the compiler variable (i.e. `CC`).

```
$ echo ${CC}

$
```

Running the SDK setup script for a 64-bit build host and an i586-tuned target architecture for a `core-image-sato` image using the current 2.5 Yocto Project release and then echoing that variable shows the value established through the script:

```
$ source /opt/poky/2.5/environment-setup-i586-poky-linux
$ echo ${CC}
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux
```

To illustrate variable use, work through this simple "Hello World!" example:

1. ***Create a Working Directory and Populate It:*** Create a clean directory for your project and then make that directory your working location.

```
$ mkdir $HOME/helloworld
$ cd $HOME/helloworld
```

After setting up the directory, populate it with files needed for the flow. You need a `main.c` file from which you call your function, a `module.h` file to contain headers, and a `module.c` that defines your function.

Create the three files as follows:

- ***main.c:***

```
#include "module.h"
void sample_func();
int main()
{
   sample_func();
   return 0;
}
```

- ***module.h:***

```
#include <stdio.h>
void sample_func();
```

- ***module.c:***

```
#include "module.h"
void sample_func()
{
        printf("Hello World!");
        printf("\n");
}
```

2. ***Source the Cross-Toolchain Environment Setup File:*** As described earlier in the manual, installing the cross-toolchain creates a cross-toolchain environment setup script in the directory that the SDK was installed. Before you can use the tools to develop your project, you must source this setup script. The script begins with the string "environment-setup" and contains the machine architecture, which is followed by the string "poky-linux". For this example, the command sources a script from the default SDK installation directory that uses the 32-bit Intel x86 Architecture and the Sumo Yocto Project release:

```
$ source /opt/poky/2.5/environment-setup-i586-poky-linux
```

3. ***Create the `Makefile`:*** For this example, the Makefile contains two lines that can be used to set the `CC` variable. One line is identical to the value that is set when you run the SDK environment setup script, and the other line sets `CC` to "gcc", the default GNU compiler on the build host:

```
# CC=i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux
# CC="gcc"
all: main.o module.o
    ${CC} main.o module.o -o target_bin
main.o: main.c module.h
        ${CC} -I . -c main.c
module.o: module.c module.h
        ${CC} -I . -c module.c
clean:
        rm -rf *.o
        rm target_bin
```

4. ***Make the Project:*** Use the `make` command to create the binary output file. Because variables are commented out in the Makefile, the value used for `CC` is the value set when the SDK environment setup file was run:

```
$ make
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux main.o module.o -o t
```

From the results of the previous command, you can see that the compiler used was the compiler established through the `CC` variable defined in the setup script.

You can override the `CC` environment variable with the same variable as set from the Makefile by uncommenting the line in the Makefile and running `make` again.

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile by uncommenting the line that sets CC to "gcc"
#
$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

As shown in the previous example, the cross-toolchain compiler is not used. Rather, the default compiler is used.

This next case shows how to override a variable by providing the variable as part of the command line. Go into the Makefile and re-insert the comment character so that running `make` uses the established SDK compiler. However, when you run `make`, use a command-line argument to set `CC` to "gcc":

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile to comment out the line setting CC to "gcc"
#
$ make
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux main.o module.o -o t
$ make clean
rm -rf *.o
rm target_bin
$ make CC="gcc"
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

In the previous case, the command-line argument overrides the SDK environment variable.

In this last case, edit Makefile again to use the "gcc" compiler but then use the "-e" option on the `make` command line:

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile to use "gcc"
#
$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
$ make clean
rm -rf *.o
rm target_bin
$ make -e
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux main.o module.o -o t
```

In the previous case, the "-e" option forces `make` to use the SDK environment variables regardless of the values in the Makefile.

5. ***Execute Your Project:*** To execute the project (i.e. `target_bin`), use the following command:

```
$ ./target_bin
Hello World!
```

> **Note**
> If you used the cross-toolchain compiler to build `target_bin` and your build host differs in architecture from that of the target machine, you need to run your project on the target device.

As expected, the project displays the "Hello World!" message.

Chapter 5. Developing Applications Using Eclipse™¶

Table of Contents

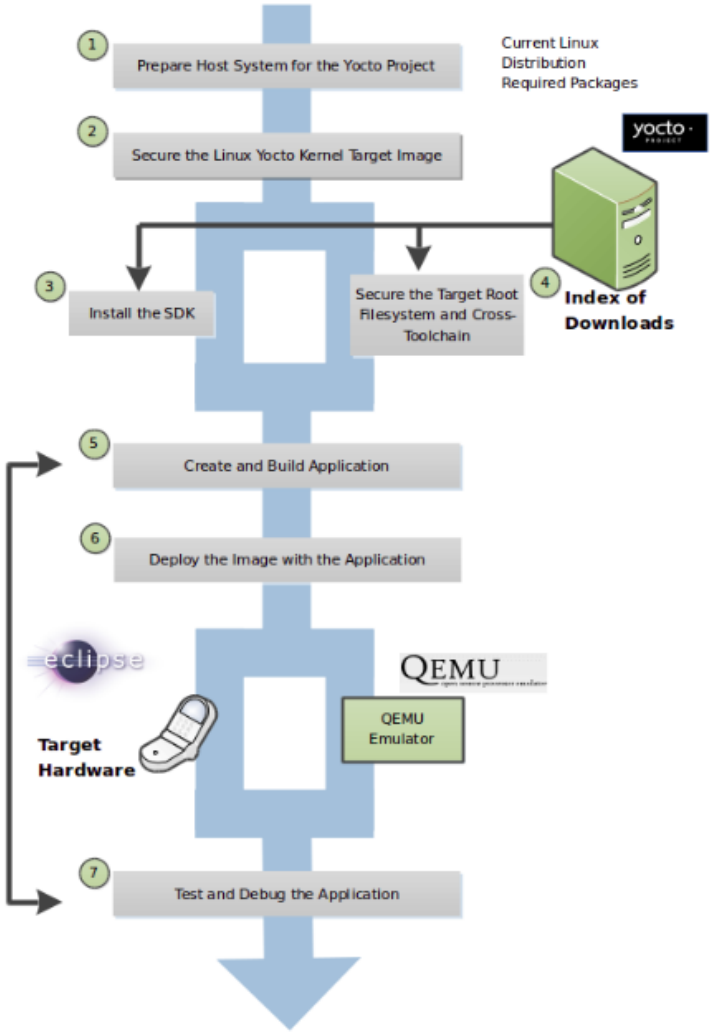
If you are familiar with the popular Eclipse IDE, you can use an Eclipse Yocto Plug-in to allow you to develop, deploy, and test your application all from within Eclipse. This chapter describes general workflow using the SDK and Eclipse and how to configure and set up Eclipse.

Notes

- This chapter assumes development of applications on top of an image prepared using the Yocto Project. As such, inclusion of a pre-built image or the building of an image is included in the workflow.
- The chapter also assumes development on a build host that is set up to use the Yocto Project. Realize that you can easily use Eclipse and the Yocto Project plug-in to develop an application for any number of images developed and tested on different machines.

5.1. Application Development Workflow Using Eclipse™¶

The following figure and supporting list summarize a general workflow for application development that uses the SDK within the Eclipse IDE. The application developed runs on top of an image created using the Yocto Project.



1. ***Prepare the Host System for the Yocto Project***: Because this example workflow assumes development on a system set up to use the Yocto Project, you need to be sure your build host can use the Yocto Project. See the "Preparing a Build Host" section in the Yocto Project Development Tasks Manual for information on how to set up your build host.

Note

Be sure you install the "xterm" package, which is a graphical and Eclipse plug-in extra needed by Eclipse.

2. ***Secure the Yocto Project Kernel Target Image***: This example workflow assumes application development on top of an image built using the Yocto Project. Depending on whether you are using a pre-built image that matches your target architecture or you are using an image you build using the OpenEmbedded Build System and where you are going to run the image while you develop your application (QEMU or real hardware), the area from which you get the image differs.
   - Download the image from `machines` if your target architecture is supported and you are going to develop and test your application on actual hardware.

- Download the image from `machines/qemu` if your target architecture is supported and you are going to develop and test your application using the QEMU Emulator.
- Build your image if you cannot find a pre-built image that matches your target architecture. If your target architecture is similar to a supported architecture, you can modify the kernel image before you build it. See the "Using `devtool` to Patch the Kernel" section in the Yocto Project Linux Kernel Development Manual for an example. You can also see the "Making a Suitable Qemux86 Image" wiki for steps needed to build an image suitable for QEMU and for debugging within the Eclipse IDE.

3. ***Install the SDK***: The SDK provides a target-specific cross-development toolchain, the root filesystem, the QEMU emulator, and other tools that can help you develop your application. For information on how to install the SDK, see the "Installing the SDK" section.
4. ***Secure the Target Root Filesystem and the Cross-Development Toolchain***: You need to find and download the appropriate root filesystem and the cross-development toolchain.

   You can find the tarballs for the root filesystem in the same area used for the kernel image. Depending on the type of image you are running, the root filesystem you need differs. For example, if you are developing an application that runs on an image that supports Sato, you need to get a root filesystem that supports Sato.

   You can find the cross-development toolchains at `toolchains`. Be sure to get the correct toolchain for your development host and your target architecture. See the "Locating Pre-Built SDK Installers" section for information and the "Installing the SDK" section for installation information.

> **Note**
> As an alternative to downloading an SDK, you can build the SDK installer. For information on building the installer, see the "Building an SDK Installer" section. Another helpful resource for building an installer is the "Cookbook guide to Making an Eclipse Debug Capable Image" wiki page.

5. ***Create and Build Your Application***: You need to have source files for your application. Once you have the files, you can use the Eclipse IDE to import them and build the project.
6. ***Deploy the Image With the Application***: Using the Eclipse IDE, you can deploy your image to the hardware or to QEMU through the project's preferences. You can also use Eclipse to load and test your image under QEMU. See the "Using the Quick EMUlator (QEMU)" chapter in the Yocto Project Development Tasks Manual for information on using QEMU.
7. ***Test and Debug the Application***: Once your application is deployed, you need to test it. Within the Eclipse IDE, you can use the debugging environment along with supported performance enhancing Linux Tools.

5.2. Working Within Eclipse¶

The Eclipse IDE is a popular development environment and it fully supports development using the Yocto Project.

When you install and configure the Eclipse Yocto Project Plug-in into the Eclipse IDE, you maximize your Yocto Project experience. Installing and configuring the Plug-in results in an environment that has extensions specifically designed to let you more easily develop software. These extensions allow for cross-compilation, deployment, and execution of your output into a QEMU emulation session as well as actual target hardware. You can also perform cross-debugging and profiling. The environment also supports performance enhancing tools that allow you to perform remote profiling, tracing, collection of power data, collection of latency data, and collection of performance data.

> **Note**
> This release of the Yocto Project supports both the Oxygen and Neon versions of the Eclipse IDE. This section provides information on how to use the Oxygen release with the Yocto Project. For information on how to use the Neon version of Eclipse with the Yocto Project, see "Appendix D".

5.2.1. Setting Up the Oxygen Version of the Eclipse IDE¶

To develop within the Eclipse IDE, you need to do the following:

1. Install the Oxygen version of the Eclipse IDE.
2. Configure the Eclipse IDE.
3. Install the Eclipse Yocto Plug-in.
4. Configure the Eclipse Yocto Plug-in.

> **Note**
> Do not install Eclipse from your distribution's package repository. Be sure to install Eclipse from the official Eclipse download site as directed in the next section.

5.2.1.1. Installing the Oxygen Eclipse IDE¶

Follow these steps to locate, install, and configure Oxygen Eclipse:

1. ***Locate the Oxygen Download:*** Open a browser and go to http://www.eclipse.org/oxygen/.
2. ***Download the Tarball:*** Click through the "Download" buttons to download the file.
3. ***Unpack the Tarball:*** Move to a clean directory and unpack the tarball. Here is an example:

   ```
   $ cd ~
   $ tar -xzvf ~/Downloads/eclipse-inst-linux64.tar.gz
   ```

   Everything unpacks into a folder named "eclipse-installer".
4. ***Launch the Installer:*** Use the following commands to launch the installer:

   ```
   $ cd ~/eclipse-installer
   $ ./eclipse-inst
   ```
5. ***Select Your IDE:*** From the list, select the "Eclipse IDE for C/C++ Developers".
6. ***Install the Software:*** Click "Install" to begin the installation. Accept all the certificates and any license agreements. Click "Install" again to finish the installation.
7. ***Launch Oxygen:*** Accept the default "workspace" and click the "Launch" button. You should see the Eclipse welcome page from which can click "workbench" to enter your workspace.

> **Note**
>
> The executable for Eclipse is located in the `eclipse/cpp-oxygen/eclipse` folder. To launch Eclipse outside of the installation process, simply execute that binary. Here is an example:
>
> ```
> $ ~/eclipse/cpp-oxygen/eclipse/eclipse
> ```

5.2.1.2. Configuring the Oxygen Eclipse IDE¶

Follow these steps to configure the Oxygen Eclipse IDE.

> **Notes**
>
> - Depending on how you installed Eclipse and what you have already done, some of the options do not appear. If you cannot find an option as directed by the manual, it has already been installed.
> - If you want to see all options regardless of whether they are installed or not, deselect the "Hide items that are already installed" check box.

1. Be sure Eclipse is running and you are in your workbench. Just click "workbench" if you are not in your default workspace.
2. Select "Install New Software" from the "Help" pull-down menu.
3. Select "Oxygen - http://download.eclipse.org/releases/oxygen" from the "Work with:" pull-down menu.
4. Expand the box next to "Linux Tools" and select the following:

   ```
   C/C++ Remote (Over TCF/TE) Run/Debug Launcher
   TM Terminal
   ```
5. Expand the box next to "Mobile and Device Development" and select the following boxes:

   ```
   C/C++ Remote (Over TCF/TE) Run/Debug Launcher
   Remote System Explorer User Actions
   TM Terminal
   TCF Remote System Explorer add-in
   TCF Target Explorer
   ```
6. Expand the box next to "Programming Languages" and select the following box:

   ```
   C/C++ Development Tools SDK
   ```
7. Complete the installation by clicking through appropriate "Next" and "Finish" buttons and then restart the Eclipse IDE.

5.2.1.3. Installing or Accessing the Oxygen Eclipse Yocto Plug-in¶

You can install the Eclipse Yocto Plug-in into the Eclipse IDE one of two ways: use the Yocto Project's Eclipse Update site to install the pre-built plug-in, or build and install the plug-in from the latest source code.

5.2.1.3.1. Installing the Pre-built Plug-in from the Yocto Project Eclipse Update Site¶

To install the Oxygen Eclipse Yocto Plug-in from the update site, follow these steps:

1. Start up the Eclipse IDE.
2. In Eclipse, select "Install New Software" from the "Help" menu.
3. Click "Add..." in the "Work with:" area.
4. Enter `http://downloads.yoctoproject.org/releases/eclipse-plugin/2.5/oxygen` in the URL field and provide a meaningful name in the "Name" field.
5. Click "OK" to have the entry automatically populate the "Work with:" field and to have the items for installation appear in the window below.
6. Check the boxes next to the following:

```
Yocto Project SDK Plug-in
Yocto Project Documentation plug-in
```

7. Complete the remaining software installation steps and then restart the Eclipse IDE to finish the installation of the plug-in.

> **Note**
> You can click "OK" when prompted about installing software that contains unsigned content.

5.2.1.3.2. Installing the Plug-in Using the Latest Source Code¶

To install the Oxygen Eclipse Yocto Plug-in from the latest source code, follow these steps:

1. Be sure your build host has JDK version 1.8 or greater. On a Linux build host you can determine the version using the following command:

```
$ java -version
```

2. Install X11-related packages:

```
$ sudo apt-get install xauth
```

3. In a new terminal shell, create a Git repository with:

```
$ cd ~
$ git clone git://git.yoctoproject.org/eclipse-yocto
```

4. Use Git to create the correct tag:

```
$ cd ~/eclipse-yocto
$ git checkout -b oxygen/sumo remotes/origin/oxygen/sumo
```

   This creates a local tag named `oxygen/sumo` based on the branch `origin/oxygen/sumo`. You are put into a detached HEAD state, which is fine since you are only going to be building and not developing.

5. Change to the `scripts` directory within the Git repository:

```
$ cd scripts
```

6. Set up the local build environment by running the setup script:

```
$ ./setup.sh
```

   When the script finishes execution, it prompts you with instructions on how to run the `build.sh` script, which is also in the `scripts` directory of the Git repository created earlier.

7. Run the `build.sh` script as directed. Be sure to provide the tag name, documentation branch, and a release name.

   Following is an example:

```
$ ECLIPSE_HOME=/home/scottrif/eclipse-yocto/scripts/eclipse ./build.sh -l oxygen/sumo master yocto-2.5 2>&1
```

   The previous example command adds the tag you need for `oxygen/sumo` to `HEAD`, then tells the build script to use the local (-l) Git checkout for the build. After running the script, the file `org.yocto.sdk-`*`release`*`-`*`date`*`-archive.zip` is in the current directory.

8. If necessary, start the Eclipse IDE and be sure you are in the Workbench.
9. Select "Install New Software" from the "Help" pull-down menu.

10. Click "Add".
11. Provide anything you want in the "Name" field.
12. Click "Archive" and browse to the ZIP file you built earlier. This ZIP file should not be "unzipped", and must be the `*archive.zip` file created by running the `build.sh` script.
13. Click the "OK" button.
14. Check the boxes that appear in the installation window to install the following:

```
Yocto Project SDK Plug-in
Yocto Project Documentation plug-in
```

15. Finish the installation by clicking through the appropriate buttons. You can click "OK" when prompted about installing software that contains unsigned content.
16. Restart the Eclipse IDE if necessary.

At this point you should be able to configure the Eclipse Yocto Plug-in as described in the "Configuring the Oxygen Eclipse Yocto Plug-in" section.

5.2.1.4. Configuring the Oxygen Eclipse Yocto Plug-In¶

Configuring the Oxygen Eclipse Yocto Plug-in involves setting the Cross Compiler options and the Target options. The configurations you choose become the default settings for all projects. You do have opportunities to change them later when you configure the project (see the following section).

To start, you need to do the following from within the Eclipse IDE:

1. Choose "Preferences" from the "Window" menu to display the Preferences Dialog.
2. Click "Yocto Project SDK" to display the configuration screen.

The following sub-sections describe how to configure the plug-in.

> **Note**
> Throughout the descriptions, a start-to-finish example for preparing a QEMU image for use with Eclipse is referenced as the "wiki" and is linked to the example on the " Cookbook guide to Making an Eclipse Debug Capable Image" wiki page.

5.2.1.4.1. Configuring the Cross-Compiler Options¶

Cross Compiler options enable Eclipse to use your specific cross compiler toolchain. To configure these options, you must select the type of toolchain, point to the toolchain, specify the sysroot location, and select the target architecture.

- ***Selecting the Toolchain Type:*** Choose between "Standalone pre-built toolchain" and "Build system derived toolchain" for Cross Compiler Options.
  - ***Standalone Pre-built Toolchain:*** Select this type when you are using a stand-alone cross-toolchain. For example, suppose you are an application developer and do not need to build a target image. Instead, you just want to use an architecture-specific toolchain on an existing kernel and target root filesystem. In other words, you have downloaded and installed a pre-built toolchain for an existing image.
  - ***Build System Derived Toolchain:*** Select this type if you built the toolchain as part of the Build Directory. When you select "Build system derived toolchain", you are using the toolchain built and bundled inside the Build Directory. For example, suppose you created a suitable image using the steps in the wiki. In this situation, you would select "Build system derived toolchain".
- ***Specify the Toolchain Root Location:*** If you are using a stand-alone pre-built toolchain, you should be pointing to where it is installed (e.g. `/opt/poky/2.5`). See the "Installing the SDK" section for information about how the SDK is installed.

  If you are using a build system derived toolchain, the path you provide for the "Toolchain Root Location" field is the Build Directory from which you run the `bitbake` command (e.g `/home/scottrif/poky/build`).

  For more information, see the "Building an SDK Installer" section.
- ***Specify Sysroot Location:*** This location is where the root filesystem for the target hardware resides.

  This location depends on where you separately extracted and installed the target filesystem when you either built it or downloaded it.

> **Note**
> If you downloaded the root filesystem for the target hardware rather than built it, you must download the `sato-sdk` image in order to build any c/c++ projects.

As an example, suppose you prepared an image using the steps in the wiki. If so, the `MY_QEMU_ROOTFS` directory is found in the Build Directory and you would browse to and select that directory (e.g. `/home/scottrif/poky/build/MY_QEMU_ROOTFS`).

For more information on how to install the toolchain and on how to extract and install the sysroot filesystem, see the "Building an SDK Installer" section.

- ***Select the Target Architecture:*** The target architecture is the type of hardware you are going to use or emulate. Use the pull-down "Target Architecture" menu to make your selection. The pull-down menu should have the supported architectures. If the architecture you need is not listed in the menu, you will need to build the image. See the "Building a Simple Image" section of the Yocto Project Development Tasks Manual for more information. You can also see the wiki.

5.2.1.4.2. Configuring the Target Options¶

You can choose to emulate hardware using the QEMU emulator, or you can choose to run your image on actual hardware.

- ***QEMU:*** Select this option if you will be using the QEMU emulator. If you are using the emulator, you also need to locate the kernel and specify any custom options.

  If you selected the Build system derived toolchain, the target kernel you built will be located in the Build Directory in `tmp/deploy/images/machine` directory. As an example, suppose you performed the steps in the wiki. In this case, you specify your Build Directory path followed by the image (e.g. `/home/scottrif/poky/build/tmp/deploy/images/qemux86/bzImage-qemux86.bin`).

  If you selected the standalone pre-built toolchain, the pre-built image you downloaded is located in the directory you specified when you downloaded the image.

  Most custom options are for advanced QEMU users to further customize their QEMU instance. These options are specified between paired angled brackets. Some options must be specified outside the brackets. In particular, the options `serial`, `nographic`, and `kvm` must all be outside the brackets. Use the `man qemu` command to get help on all the options and their use. The following is an example:

  ```
  serial ‘<-m 256 -full-screen>’
  ```

  Regardless of the mode, Sysroot is already defined as part of the Cross-Compiler Options configuration in the "Sysroot Location:" field.

- ***External HW:*** Select this option if you will be using actual hardware.

Click "Apply and Close" to save your plug-in configurations.

5.2.2. Creating the Project¶

You can create two types of projects: Autotools-based, or Makefile-based. This section describes how to create Autotools-based projects from within the Eclipse IDE. For information on creating Makefile-based projects in a terminal window, see the "Makefile-Based Projects" section.

> **Note**
> Do not use special characters in project names (e.g. spaces, underscores, etc.). Doing so can cause configuration to fail.

To create a project based on a Yocto template and then display the source code, follow these steps:

1. Select "C/C++ Project" from the "File -> New" menu.
2. Select "C Managed Build" from the available options and click "Next".
3. Expand "Yocto Project SDK Autotools Project".
4. Select "Hello World ANSI C Autotools Projects". This is an Autotools-based project based on a Yocto template.
5. Put a name in the "Project name:" field. Do not use hyphens as part of the name (e.g. "hello").
6. Click "Next".
7. Add appropriate information in the various fields.
8. Click "Finish".
9. If the "open perspective" prompt appears, click "Yes" so that you in the C/C++ perspective.
10. The left-hand navigation pane shows your project. You can display your source by double clicking the project's source file.

5.2.3. Configuring the Cross-Toolchains¶

The earlier section, "Configuring the Oxygen Eclipse Yocto Plug-in", sets up the default project configurations. You can override these settings for a given project by following these steps:

1. Select "Yocto Project Settings" from the "Project -> Properties" menu. This selection brings up the Yocto Project Settings Dialog and allows you to make changes specific to an individual project.

   By default, the Cross Compiler Options and Target Options for a project are inherited from settings you provided using the Preferences Dialog as described earlier in the "Configuring the Oxygen Eclipse Yocto Plug-in" section. The Yocto Project Settings Dialog allows you to override those default settings for a given project.

2. Make or verify your configurations for the project and click "Apply and Close".
3. Right-click in the navigation pane and select "Reconfigure Project" from the pop-up menu. This selection reconfigures the project by running Autotools GNU utility programs such as Autoconf, Automake, and so forth in the workspace for your project. Click on the "Console" tab beneath your source code to see the results of reconfiguring your project.

5.2.4. Building the Project¶

To build the project select "Build All" from the "Project" menu. The console should update and you can note the cross-compiler you are using (i.e. `i586-poky-linux-gcc` in this example).

> **Note**
> When building "Yocto Project SDK Autotools" projects, the Eclipse IDE might display error messages for Functions/Symbols/Types that cannot be "resolved", even when the related include file is listed at the project navigator and when the project is able to build. For these cases only, it is recommended to add a new linked folder to the appropriate sysroot. Use these steps to add the linked folder:
>
> 1. Select the project.
> 2. Select "Folder" from the "File -> New" menu.
> 3. In the "New Folder" Dialog, click the "Advanced" button and then activate "Link to alternate location (linked folder)" button.
> 4. Click "Browse" to navigate to the include folder inside the same sysroot location selected in the Yocto Project configuration preferences.
> 5. Click "Finish" to save the linked folder.

5.2.5. Starting QEMU in User-Space NFS Mode¶

To start the QEMU emulator from within Eclipse, follow these steps:

> **Note**
> See the "Using the Quick EMUlator (QEMU)" chapter in the Yocto Project Development Tasks Manual for more information on using QEMU.

1. Expose and select "External Tools Configurations ..." from the "Run -> External Tools" menu.
2. Locate and select your image in the navigation panel to the left (e.g. `qemu_i586-poky-linux`).
3. Click "Run" to launch QEMU.

> **Note**
> The host on which you are running QEMU must have the `rpcbind` utility running to be able to make RPC calls on a server on that machine. If QEMU does not invoke and you receive error messages involving `rpcbind`, follow the suggestions to get the service running. As an example, on a new Ubuntu 16.04 LTS installation, you must do the following in a new shell in order to get QEMU to launch:
>
> ```
> $ sudo apt-get install rpcbind
> ```
>
> After installing `rpcbind`, you need to edit the `/etc/init.d/rpcbind` file to include the following line:
>
> ```
> OPTIONS="-i -w"
> ```
>
> After modifying the file, you need to start the service:
>
> ```
> $ sudo service portmap restart
> ```

4. If needed, enter your host root password in the shell window at the prompt. This sets up a `Tap 0` connection needed for running in user-space NFS mode.
5. Wait for QEMU to launch.
6. Once QEMU launches, you can begin operating within that environment. One useful task at this point would be to determine the IP Address for the user-space NFS by using the `ifconfig` command. The IP address of the QEMU machine appears in the xterm window. You can use this address to help you see which particular IP address the instance of QEMU is using.

5.2.6. Deploying and Debugging the Application¶

Once the QEMU emulator is running the image, you can deploy your application using the Eclipse IDE and then use the emulator to perform debugging. Follow these steps to deploy the application.

> **Note**
> Currently, Eclipse does not support SSH port forwarding. Consequently, if you need to run or debug a remote application using the host display, you must create a tunneling connection from outside Eclipse and keep that connection alive during your work. For example, in a new terminal, run the following:
>
> ```
> $ ssh -XY user_name@remote_host_ip
> ```
>
> Using the above form, here is an example:
>
> ```
> $ ssh -XY root@192.168.7.2
> ```
>
> After running the command, add the command to be executed in Eclipse's run configuration before the application as follows:
>
> ```
> export DISPLAY=:10.0
> ```
>
> Be sure to not destroy the connection during your QEMU session (i.e. do not exit out of or close that shell).

1. Select "Debug Configurations..." from the "Run" menu.
2. In the left area, expand "C/C++Remote Application".
3. Locate your project and select it to bring up a new tabbed view in the Debug Configurations Dialog.
4. Click on the "Debugger" tab to see the cross-tool debugger you are using. Be sure to change to the debugger perspective in Eclipse.
5. Click on the "Main" tab.
6. Create a new connection to the QEMU instance by clicking on "new".
7. Select "SSH", which means Secure Socket Shell and then click "OK". Optionally, you can select a TCF connection instead.
8. Clear out the "Connection name" field and enter any name you want for the connection.
9. Put the IP address for the connection in the "Host" field. For QEMU, the default is "192.168.7.2". However, if a previous QEMU session did not exit cleanly, the IP address increments (e.g. "192.168.7.3").

> **Note**
> You can find the IP address for the current QEMU session by looking in the xterm that opens when you launch QEMU.

10. Enter "root", which is the default for QEMU, for the "User" field. Be sure to leave the password field empty.
11. Click "Finish" to close the New Connections Dialog.
12. If necessary, use the drop-down menu now in the "Connection" field and pick the IP Address you entered.
13. Assuming you are connecting as the root user, which is the default for QEMU x86-64 SDK images provided by the Yocto Project, in the "Remote Absolute File Path for C/C++ Application" field, browse to `/home/root/`*`ProjectName`* (e.g. `/home/root/hello`). You could also browse to any other path you have write access to on the target such as `/usr/bin`. This location is where your application will be located on the QEMU system. If you fail to browse to and specify an appropriate location, QEMU will not understand what to remotely launch. Eclipse is helpful in that it auto fills your application name for you assuming you browsed to a directory.

> **Tips**
> - If you are prompted to provide a username and to optionally set a password, be sure you provide "root" as the username and you leave the password field blank.
> - If browsing to a directory fails or times out, but you can `ssh` into your QEMU or target from the command line and you have proxies set up, it is likely that Eclipse is sending the SSH traffic to a proxy. In this case, either use TCF , or click on "Configure proxy settings" in the connection dialog and add the target IP address to the "bypass proxy" section. You might also need to change "Active Provider" from Native to Manual.

14. Be sure you change to the "Debug" perspective in Eclipse.
15. Click "Debug"

16. Accept the debug perspective.

5.2.7. Using Linuxtools¶

As mentioned earlier in the manual, performance tools exist (Linuxtools) that enhance your development experience. These tools are aids in developing and debugging applications and images. You can run these tools from within the Eclipse IDE through the "Linuxtools" menu.

For information on how to configure and use these tools, see http://www.eclipse.org/linuxtools/.

Appendix A. Obtaining the SDK¶

Table of Contents

A.1. Locating Pre-Built SDK Installers¶

You can use existing, pre-built toolchains by locating and running an SDK installer script that ships with the Yocto Project. Using this method, you select and download an architecture-specific SDK installer and then run the script to hand-install the toolchain.

Follow these steps to locate and hand-install the toolchain:

1. ***Go to the Installers Directory:*** Go to http://downloads.yoctoproject.org/releases/yocto/yocto-2.5/toolchain/
2. ***Open the Folder for Your Build Host:*** Open the folder that matches your build host (i.e. `i686` for 32-bit machines or `x86_64` for 64-bit machines).
3. ***Locate and Download the SDK Installer:*** You need to find and download the installer appropriate for your build host, target hardware, and image type.

   The installer files (`*.sh`) follow this naming convention:

   ```
   poky-glibc-host_system-core-image-type-arch-toolchain[-ext]-release.sh

   Where:
       host_system is a string representing your development system:
              "i686" or "x86_64"

       type is a string representing the image:
              "sato" or "minimal"

       arch is a string representing the target architecture:
              "aarch64", "armv5e", "core2-64", "coretexa8hf-neon", "i586", "mips32r2",
              "mips64", or "ppc7400"

       release is the version of Yocto Project.

       NOTE:
          The standard SDK installer does not have the "-ext" string as
          part of the filename.
   ```

   The toolchains provided by the Yocto Project are based off of the `core-image-sato` and `core-image-minimal` images and contain libraries appropriate for developing against those images.

   For example, if your build host is a 64-bit x86 system and you need an extended SDK for a 64-bit core2 target, go into the `x86_64` folder and download the following installer:

   ```
   poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-2.5.sh
   ```

4. ***Run the Installer:*** Be sure you have execution privileges and run the installer. Following is an example from the `Downloads` directory:

   ```
   $ ~/Downloads/poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-2.5.sh
   ```

   During execution of the script, you choose the root location for the toolchain. See the "Installed Standard SDK Directory Structure" section and the "Installed Extensible SDK Directory Structure" section for more information.

A.2. Building an SDK Installer¶

As an alternative to locating and downloading an SDK installer, you can build the SDK installer. Follow these steps:

1. ***Set Up the Build Environment:*** Be sure you are set up to use BitBake in a shell. See the "Preparing the Build Host" section in the Yocto Project Development Tasks Manual for information on how to get a build host ready that is either a native Linux machine or a machine that uses CROPS.

2. ***Clone the `poky` Repository:*** You need to have a local copy of the Yocto Project Source Directory (i.e. a local `poky` repository). See the "Cloning the `poky` Repository" and possibly the "Checking Out by Branch in Poky" and "Checking Out by Tag in Poky" sections all in the Yocto Project Development Tasks Manual for information on how to clone the `poky` repository and check out the appropriate branch for your work.

3. ***Initialize the Build Environment:*** While in the root directory of the Source Directory (i.e. `poky`), run the `oe-init-build-env` environment setup script to define the OpenEmbedded build environment on your build host.

   ```
   $ source oe-init-build-env
   ```

   Among other things, the script creates the Build Directory, which is `build` in this case and is located in the Source Directory. After the script runs, your current working directory is set to the `build` directory.

4. ***Make Sure You Are Building an Installer for the Correct Machine:*** Check to be sure that your `MACHINE` variable in the `local.conf` file in your Build Directory matches the architecture for which you are building.

5. ***Make Sure Your SDK Machine is Correctly Set:*** If you are building a toolchain designed to run on an architecture that differs from your current development host machine (i.e. the build host), be sure that the `SDKMACHINE` variable in the `local.conf` file in your Build Directory is correctly set.

   ### Note

   If you are building an SDK installer for the Extensible SDK, the `SDKMACHINE` value must be set for the architecture of the machine you are using to build the installer. If `SDKMACHINE` is not set appropriately, the build fails and provides an error message similar to the following:

   ```
   The extensible SDK can currently only be built for the same architecture as the mac
   set to i686 (likely via setting SDKMACHINE) which is different from the architectur
   Unable to continue.
   ```

6. ***Build the SDK Installer:*** To build the SDK installer for a standard SDK and populate the SDK image, use the following command form. Be sure to replace *`image`* with an image (e.g. "core-image-sato"):

   ```
   $ bitbake image -c populate_sdk
   ```

   You can do the same for the extensible SDK using this command form:

   ```
   $ bitbake image -c populate_sdk_ext
   ```

   These commands produce an SDK installer that contains the sysroot that matches your target root filesystem.

   When the `bitbake` command completes, the SDK installer will be in `tmp/deploy/sdk` in the Build Directory.

   ### Notes

   - By default, the previous BitBake command does not build static binaries. If you want to use the toolchain to build these types of libraries, you need to be sure your SDK has the appropriate static development libraries. Use the `TOOLCHAIN_TARGET_TASK` variable inside your `local.conf` file before building the SDK installer. Doing so ensures that the eventual SDK installation process installs the appropriate library packages as part of the SDK. Following is an example using `libc` static development libraries:

     ```
     TOOLCHAIN_TARGET_TASK_append = " libc-staticdev"
     ```

   - For additional information on building the installer, see the Cookbook guide to Making an Eclipse™ Debug Capable Image wiki page.

7. ***Run the Installer:*** You can now run the SDK installer from `tmp/deploy/sdk` in the Build Directory. Following is an example:

   ```
   $ cd ~/poky/build/tmp/deploy/sdk
   $ ./poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-2.5.sh
   ```

   During execution of the script, you choose the root location for the toolchain. See the "Installed Standard SDK Directory Structure" section and the "Installed Extensible SDK Directory Structure" section for more information.

A.3. Extracting the Root Filesystem¶

After installing the toolchain, for some use cases you might need to separately extract a root filesystem:

- You want to boot the image using NFS.
- You want to use the root filesystem as the target sysroot. For example, the Eclipse IDE environment with the Eclipse Yocto Plug-in installed allows you to use QEMU to boot under NFS.
- You want to develop your target application using the root filesystem as the target sysroot.

Follow these steps to extract the root filesystem:

1. ***Locate and Download the Tarball for the Pre-Built Root Filesystem Image File:*** You need to find and download the root filesystem image file that is appropriate for your target system. These files are kept in machine-specific folders in the Index of Releases in the "machines" directory.

   The machine-specific folders of the "machines" directory contain tarballs (`*.tar.bz2`) for supported machines. These directories also contain flattened root filesystem image files (`*.ext4`), which you can use with QEMU directly.

   The pre-built root filesystem image files follow these naming conventions:

   ```
   core-image-profile-arch.tar.bz2

   Where:
       profile is the filesystem image's profile:
                 lsb, lsb-dev, lsb-sdk, minimal, minimal-dev, minimal-initramfs,
                 sato, sato-dev, sato-sdk, sato-sdk-ptest. For information on
                 these types of image profiles, see the "Images" chapter in
                 the Yocto Project Reference Manual.

       arch is a string representing the target architecture:
                 beaglebone-yocto, beaglebone-yocto-lsb, edgerouter, edgerouter-lsb,
                 genericx86, genericx86-64, genericx86-64-lsb, genericx86-lsb,
                 mpc8315e-rdb, mpc8315e-rdb-lsb, and qemu*.
   ```

   The root filesystems provided by the Yocto Project are based off of the `core-image-sato` and `core-image-minimal` images.

   For example, if you plan on using a BeagleBone device as your target hardware and your image is a `core-image-sato-sdk` image, you can download the following file:

   ```
   core-image-sato-sdk-beaglebone-yocto.tar.bz2
   ```

2. ***Initialize the Cross-Development Environment:*** You must `source` the cross-development environment setup script to establish necessary environment variables.

   This script is located in the top-level directory in which you installed the toolchain (e.g. `poky_sdk`).

   Following is an example based on the toolchain installed in the "Locating Pre-Built SDK Installers" section:

   ```
   $ source ~/poky_sdk/environment-setup-core2-64-poky-linux
   ```

3. ***Extract the Root Filesystem:*** Use the `runqemu-extract-sdk` command and provide the root filesystem image.

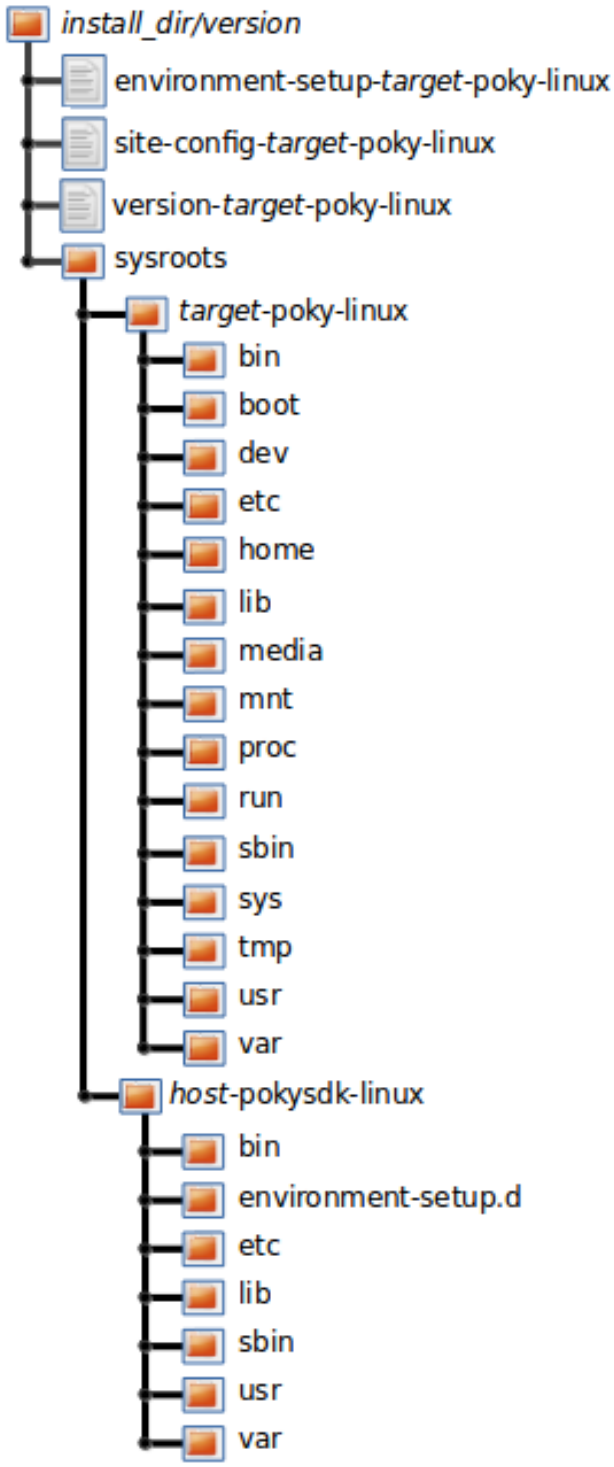
   Following is an example command that extracts the root filesystem from a previously built root filesystem image that was downloaded from the Index of Releases. This command extracts the root filesystem into the `core2-64-sato` directory:

   ```
   $ runqemu-extract-sdk ~/Downloads/core-image-sato-sdk-beaglebone-yocto.tar.bz2 ~/beaglebone-sato
   ```

   You could now point to the target sysroot at `beablebone-sato`.

A.4. Installed Standard SDK Directory Structure¶

The following figure shows the resulting directory structure after you install the Standard SDK by running the `*.sh` SDK installation script:

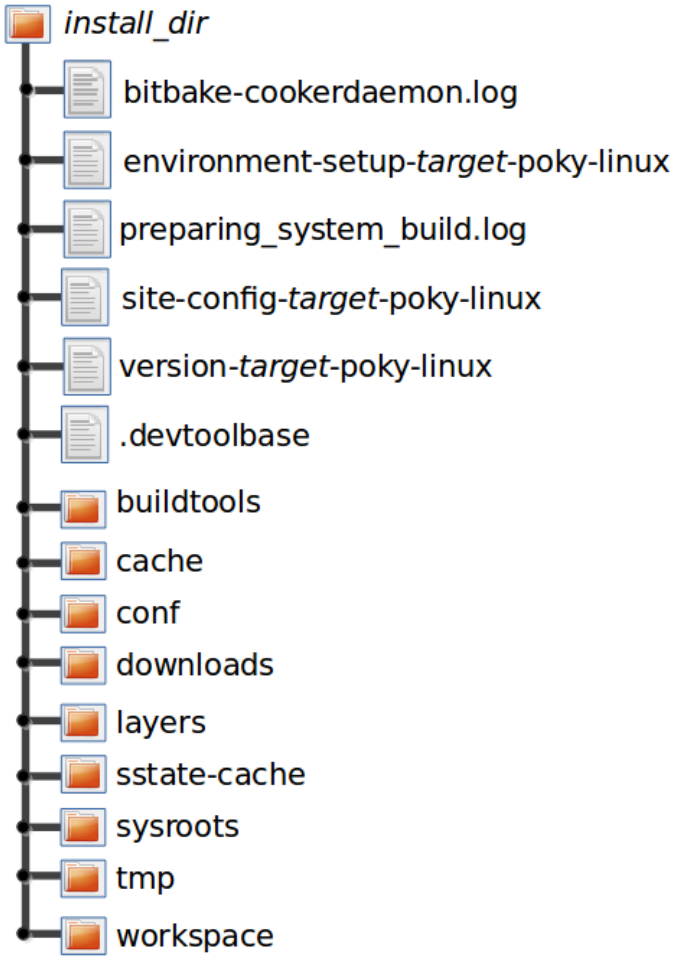


The installed SDK consists of an environment setup script for the SDK, a configuration file for the target, a version file for the target, and the root filesystem (`sysroots`) needed to develop objects for the target system.

Within the figure, italicized text is used to indicate replaceable portions of the file or directory name. For example, *`install_dir/version`* is the directory where the SDK is installed. By default, this directory is `/opt/poky/`. And, *`version`* represents the specific snapshot of the SDK (e.g. `2.5`). Furthermore, *`target`* represents the target architecture (e.g. `i586`) and *`host`* represents the development system's architecture (e.g. `x86_64`). Thus, the complete names of the two directories within the `sysroots` could be `i586-poky-linux` and `x86_64-pokysdk-linux` for the target and host, respectively.

A.5. Installed Extensible SDK Directory Structure¶

The following figure shows the resulting directory structure after you install the Extensible SDK by running the `*.sh` SDK installation script:



The installed directory structure for the extensible SDK is quite different than the installed structure for the standard SDK. The extensible SDK does not separate host and target parts in the same manner as does the standard SDK. The extensible SDK uses an embedded copy of the OpenEmbedded build system, which has its own sysroots.

Of note in the directory structure are an environment setup script for the SDK, a configuration file for the target, a version file for the target, and log files for the OpenEmbedded build system preparation script run by the installer and BitBake.

Within the figure, italicized text is used to indicate replaceable portions of the file or directory name. For example, *install_dir* is the directory where the SDK is installed, which is `poky_sdk` by default, and *target* represents the target architecture (e.g. `i586`).

Appendix B. Customizing the Extensible SDK¶

Table of Contents

This appendix presents customizations you can apply to the extensible SDK.

B.1. Configuring the Extensible SDK¶

The extensible SDK primarily consists of a pre-configured copy of the OpenEmbedded build system from which it was produced. Thus, the SDK's configuration is derived using that build system and the filters shown in the following list. When these filters are present, the OpenEmbedded build system applies them against `local.conf` and `auto.conf`:

- Variables whose values start with "/" are excluded since the assumption is that those values are paths that are likely to be specific to the build host.
- Variables listed in `SDK_LOCAL_CONF_BLACKLIST` are excluded. These variables are not allowed through from the OpenEmbedded build system configuration into the extensible SDK configuration. Typically, these variables are specific to the machine on which the build system is running and could be problematic as part of the extensible SDK configuration.

  For a list of the variables excluded by default, see the `SDK_LOCAL_CONF_BLACKLIST` in the glossary of the Yocto Project Reference Manual.

- Variables listed in SDK_LOCAL_CONF_WHITELIST are included. Including a variable in the value of SDK_LOCAL_CONF_WHITELIST overrides either of the previous two filters. The default value is blank.
- Classes inherited globally with INHERIT that are listed in SDK_INHERIT_BLACKLIST are disabled. Using SDK_INHERIT_BLACKLIST to disable these classes is the typical method to disable classes that are problematic or unnecessary in the SDK context. The default value blacklists the buildhistory and icecc classes.

Additionally, the contents of conf/sdk-extra.conf, when present, are appended to the end of conf/local.conf within the produced SDK, without any filtering. The sdk-extra.conf file is particularly useful if you want to set a variable value just for the SDK and not the OpenEmbedded build system used to create the SDK.

B.2. Adjusting the Extensible SDK to Suit Your Build Host's Setup¶

In most cases, the extensible SDK defaults should work with your build host's setup. However, some cases exist for which you might consider making adjustments:

- If your SDK configuration inherits additional classes using the INHERIT variable and you do not need or want those classes enabled in the SDK, you can blacklist them by adding them to the SDK_INHERIT_BLACKLIST variable as described in the fourth bullet of the previous section.

> **Note**
> The default value of SDK_INHERIT_BLACKLIST is set using the "?=" operator. Consequently, you will need to either define the entire list by using the "=" operator, or you will need to append a value using either "_append" or the "+=" operator. You can learn more about these operators in the "Basic Syntax" section of the BitBake User Manual.

.

- If you have classes or recipes that add additional tasks to the standard build flow (i.e. the tasks execute as the recipe builds as opposed to being called explicitly), then you need to do one of the following:
  - After ensuring the tasks are shared state tasks (i.e. the output of the task is saved to and can be restored from the shared state cache) or ensuring the tasks are able to be produced quickly from a task that is a shared state task, add the task name to the value of SDK_RECRDEP_TASKS.
  - Disable the tasks if they are added by a class and you do not need the functionality the class provides in the extensible SDK. To disable the tasks, add the class to the SDK_INHERIT_BLACKLIST variable as described in the previous section.
- Generally, you want to have a shared state mirror set up so users of the SDK can add additional items to the SDK after installation without needing to build the items from source. See the "Providing Additional Installable Extensible SDK Content" section for information.
- If you want users of the SDK to be able to easily update the SDK, you need to set the SDK_UPDATE_URL variable. For more information, see the "Providing Updates to the Extensible SDK After Installation" section.
- If you have adjusted the list of files and directories that appear in COREBASE (other than layers that are enabled through bblayers.conf), then you must list these files in COREBASE_FILES so that the files are copied into the SDK.
- If your OpenEmbedded build system setup uses a different environment setup script other than oe-init-build-env, then you must set OE_INIT_ENV_SCRIPT to point to the environment setup script you use.

> **Note**
> You must also reflect this change in the value used for the COREBASE_FILES variable as previously described.

B.3. Changing the Extensible SDK Installer Title¶

You can change the displayed title for the SDK installer by setting the SDK_TITLE variable. By default, this title is derived from DISTRO_NAME when it is set. If the DISTRO_NAME variable is not set, the title is derived from the DISTRO variable.

The populate_sdk_ext class defines the default value of the SDK_TITLE variable as follows:

```
SDK_TITLE_task-populate-sdk-ext = "${@d.getVar('DISTRO_NAME') or d.getVar('DISTRO')} Extensible SDK"
```

B.4. Providing Updates to the Extensible SDK After Installation¶

When you make changes to your configuration or to the metadata and if you want those changes to be reflected in installed SDKs, you need to perform additional steps. These steps make it possible for anyone using the installed SDKs to update the installed SDKs by using the devtool sdk-update command:

1. Create a directory that can be shared over HTTP or HTTPS. You can do this by setting up a web server such as an Apache HTTP Server or Nginx server in the cloud to host the directory. This directory must contain the published SDK.
2. Set the SDK_UPDATE_URL variable to point to the corresponding HTTP or HTTPS URL. Setting this variable causes any SDK built to default to that URL and thus, the user does not have to pass the URL to the devtool sdk-update command as described in the "Applying Updates to an Installed Extensible SDK" section.
3. Build the extensible SDK normally (i.e., use the bitbake -c populate_sdk_ext *imagename* command).
4. Publish the SDK using the following command:

   ```
   $ oe-publish-sdk some_path/sdk-installer.sh path_to_shared_http_directory
   ```

   You must repeat this step each time you rebuild the SDK with changes that you want to make available through the update mechanism.

Completing the above steps allows users of the existing installed SDKs to simply run devtool sdk-update to retrieve and apply the latest updates. See the "Applying Updates to an Installed Extensible SDK" section for further information.

B.5. Providing Additional Installable Extensible SDK Content¶

If you want the users of an extensible SDK you build to be able to add items to the SDK without requiring the users to build the items from source, you need to do a number of things:

1. Ensure the additional items you want the user to be able to install are already built:
   - Build the items explicitly. You could use one or more "meta" recipes that depend on lists of other recipes.
   - Build the "world" target and set EXCLUDE_FROM_WORLD_pn-*recipename* for the recipes you do not want built. See the EXCLUDE_FROM_WORLD variable for additional information.
2. Expose the sstate-cache directory produced by the build. Typically, you expose this directory by making it available through an Apache HTTP Server or Nginx server.
3. Set the appropriate configuration so that the produced SDK knows how to find the configuration. The variable you need to set is SSTATE_MIRRORS:

   ```
   SSTATE_MIRRORS = "file://.*  http://example.com/some_path/sstate-cache/PATH"
   ```

   You can set the SSTATE_MIRRORS variable in two different places:

   - If the mirror value you are setting is appropriate to be set for both the OpenEmbedded build system that is actually building the SDK and the SDK itself (i.e. the mirror is accessible in both places or it will fail quickly on the OpenEmbedded build system side, and its contents will not interfere with the build), then you can set the variable in your local.conf or custom distro configuration file. You can then "whitelist" the variable through to the SDK by adding the following:

     ```
     SDK_LOCAL_CONF_WHITELIST = "SSTATE_MIRRORS"
     ```

   - Alternatively, if you just want to set the SSTATE_MIRRORS variable's value for the SDK alone, create a conf/sdk-extra.conf file either in your Build Directory or within any layer and put your SSTATE_MIRRORS setting within that file.

> **Note**
> This second option is the safest option should you have any doubts as to which method to use when setting SSTATE_MIRRORS.

B.6. Minimizing the Size of the Extensible SDK Installer Download¶

By default, the extensible SDK bundles the shared state artifacts for everything needed to reconstruct the image for which the SDK was built. This bundling can lead to an SDK installer file that is a Gigabyte or more in size. If the size of this file causes a problem, you can build an SDK that has just enough in it to install and provide access to the devtool command by setting the following in your configuration:

```
SDK_EXT_TYPE = "minimal"
```

Setting SDK_EXT_TYPE to "minimal" produces an SDK installer that is around 35 Mbytes in size, which downloads and installs quickly. You need to realize, though, that the minimal installer does not install any libraries or tools out of the box. These libraries and tools must be installed either "on the fly" or through actions you perform using devtool or explicitly with the devtool sdk-install command.

In most cases, when building a minimal SDK you need to also enable bringing in the information on a wider range of packages produced by the system. Requiring this wider range of information is particularly true so that devtool add is able to effectively map dependencies it discovers in a source tree to the appropriate recipes. Additionally, the information enables the devtool search command to return useful results.

To facilitate this wider range of information, you would need to set the following:

```
SDK_INCLUDE_PKGDATA = "1"
```

See the SDK_INCLUDE_PKGDATA variable for additional information.

Setting the SDK_INCLUDE_PKGDATA variable as shown causes the "world" target to be built so that information for all of the recipes included within it are available. Having these recipes available increases build time significantly and increases the size of the SDK installer by 30-80 Mbytes depending on how many recipes are included in your configuration.

You can use EXCLUDE_FROM_WORLD_pn-*recipename* for recipes you want to exclude. However, it is assumed that you would need to be building the "world" target if you want to provide additional items to the SDK. Consequently, building for "world" should not represent undue overhead in most cases.

> **Note**
> If you set SDK_EXT_TYPE to "minimal", then providing a shared state mirror is mandatory so that items can be installed as needed. See the "Providing Additional Installable Extensible SDK Content" section for more information.

You can explicitly control whether or not to include the toolchain when you build an SDK by setting the SDK_INCLUDE_TOOLCHAIN variable to "1". In particular, it is useful to include the toolchain when you have set SDK_EXT_TYPE to "minimal", which by default, excludes the toolchain. Also, it is helpful if you are building a small SDK for use with an IDE, such as Eclipse™, or some other tool where you do not want to take extra steps to install a toolchain.

Appendix C. Customizing the Standard SDK¶

Table of Contents

This appendix presents customizations you can apply to the standard SDK.

C.1. Adding Individual Packages to the Standard SDK¶

When you build a standard SDK using the bitbake -c populate_sdk, a default set of packages is included in the resulting SDK. The TOOLCHAIN_HOST_TASK and TOOLCHAIN_TARGET_TASK variables control the set of packages adding to the SDK.

If you want to add individual packages to the toolchain that runs on the host, simply add those packages to the TOOLCHAIN_HOST_TASK variable. Similarly, if you want to add packages to the default set that is part of the toolchain that runs on the target, add the packages to the TOOLCHAIN_TARGET_TASK variable.

C.2. Adding API Documentation to the Standard SDK¶

You can include API documentation as well as any other documentation provided by recipes with the standard SDK by adding "api-documentation" to the DISTRO_FEATURES variable:

```
DISTRO_FEATURES_append = " api-documentation"
```

Setting this variable as shown here causes the OpenEmbedded build system to build the documentation and then include it in the standard SDK.

Appendix D. Using Eclipse™ Neon¶

Table of Contents

This release of the Yocto Project supports both the Oxygen and Neon versions of the Eclipse IDE. This appendix presents information that describes how to obtain and configure the Neon version of Eclipse. It also provides a basic project example that

you can work through from start to finish. For general information on using the Eclipse IDE and the Yocto Project Eclipse Plug-In, see the "Application Development Workflow Using Eclipse™" section.

D.1. Setting Up the Neon Version of the Eclipse IDE¶

To develop within the Eclipse IDE, you need to do the following:

1. Install the Neon version of the Eclipse IDE.
2. Configure the Eclipse IDE.
3. Install the Eclipse Yocto Plug-in.
4. Configure the Eclipse Yocto Plug-in.

Note

Do not install Eclipse from your distribution's package repository. Be sure to install Eclipse from the official Eclipse download site as directed in the next section.

D.1.1. Installing the Neon Eclipse IDE¶

Follow these steps to locate, install, and configure Neon Eclipse:

1. ***Locate the Neon Download:*** Open a browser and go to http://www.eclipse.org/neon/.
2. ***Download the Tarball:*** Click the "Download" button and look for the "Eclipse IDE for C/C++ Developers" Neon 3 Package. Select the correct platform download link listed at the right. For example, click on "64-bit" next to Linux if your build host is running a 64-bit Linux distribution. Click through the process to save the file.
3. ***Unpack the Tarball:*** Move to a directory and unpack the tarball. The following commands unpack the tarball into the home directory:

   ```
   $ cd ~
   $ tar -xzvf ~/Downloads/eclipse-cpp-neon-3-linux-gtk-x86_64.tar.gz
   ```

   Everything unpacks into a folder named "Eclipse".
4. ***Launch Eclipse:*** The following commands launch Eclipse assuming you unpacked it in your home directory:

   ```
   $ cd ~/eclipse
   $ ./eclipse
   ```

   Accept the default "workspace" once Eclipse launches.

D.1.2. Configuring the Neon Eclipse IDE¶

Follow these steps to configure the Neon Eclipse IDE.

Notes

- Depending on how you installed Eclipse and what you have already done, some of the options do not appear. If you cannot find an option as directed by the manual, it has already been installed.
- If you want to see all options regardless of whether they are installed or not, deselect the "Hide items that are already installed" check box.

1. Be sure Eclipse is running and you are in your workbench.
2. Select "Install New Software" from the "Help" pull-down menu.
3. Select "Neon - http://download.eclipse.org/releases/neon" from the "Work with:" pull-down menu.
4. Expand the box next to "Linux Tools" and select the following

   ```
   C/C++ Remote (Over TCF/TE) Run/Debug Launcher
   TM Terminal
   ```

5. Expand the box next to "Mobile and Device Development" and select the following boxes:

   ```
   C/C++ Remote (Over TCF/TE) Run/Debug Launcher
   Remote System Explorer User Actions
   TM Terminal
   TCF Remote System Explorer add-in
   TCF Target Explorer
   ```

6. Expand the box next to "Programming Languages" and select the following box:

```
C/C++ Development Tools SDK
```

7. Complete the installation by clicking through appropriate "Next" and "Finish" buttons.

D.1.3. Installing or Accessing the Neon Eclipse Yocto Plug-in¶

You can install the Eclipse Yocto Plug-in into the Eclipse IDE one of two ways: use the Yocto Project's Eclipse Update site to install the pre-built plug-in or build and install the plug-in from the latest source code.

D.1.3.1. Installing the Pre-built Plug-in from the Yocto Project Eclipse Update Site¶

To install the Neon Eclipse Yocto Plug-in from the update site, follow these steps:

1. Start up the Eclipse IDE.
2. In Eclipse, select "Install New Software" from the "Help" menu.
3. Click "Add..." in the "Work with:" area.
4. Enter `http://downloads.yoctoproject.org/releases/eclipse-plugin/2.5/neon` in the URL field and provide a meaningful name in the "Name" field.
5. Click "OK" to have the entry automatically populate the "Work with:" field and to have the items for installation appear in the window below.
6. Check the boxes next to the following:

   ```
   Yocto Project SDK Plug-in
   Yocto Project Documentation plug-in
   ```

7. Complete the remaining software installation steps and then restart the Eclipse IDE to finish the installation of the plug-in.

> **Note**
> You can click "OK" when prompted about installing software that contains unsigned content.

D.1.3.2. Installing the Plug-in Using the Latest Source Code¶

To install the Neon Eclipse Yocto Plug-in from the latest source code, follow these steps:

1. Be sure your build host has JDK version 1.8 or greater. On a Linux build host you can determine the version using the following command:

   ```
   $ java -version
   ```

2. install X11-related packages:

   ```
   $ sudo apt-get install xauth
   ```

3. In a new terminal shell, create a Git repository with:

   ```
   $ cd ~
   $ git clone git://git.yoctoproject.org/eclipse-yocto
   ```

4. Use Git to create the correct tag:

   ```
   $ cd ~/eclipse-yocto
   $ git checkout -b neon/sumo remotes/origin/neon/sumo
   ```

   This creates a local tag named `neon/sumo` based on the branch `origin/neon/sumo`. You are put into a detached HEAD state, which is fine since you are only going to be building and not developing.

5. Change to the `scripts` directory within the Git repository:

   ```
   $ cd scripts
   ```

6. Set up the local build environment by running the setup script:

   ```
   $ ./setup.sh
   ```

   When the script finishes execution, it prompts you with instructions on how to run the `build.sh` script, which is also in the `scripts` directory of the Git repository created earlier.

7. Run the `build.sh` script as directed. Be sure to provide the tag name, documentation branch, and a release name.

   Following is an example:

   ```
   $ ECLIPSE_HOME=/home/scottrif/eclipse-yocto/scripts/eclipse ./build.sh -l neon/sumo master yocto-2.5 2>&1 |
   ```

The previous example command adds the tag you need for `neon/sumo` to `HEAD`, then tells the build script to use the local (-l) Git checkout for the build. After running the script, the file `org.yocto.sdk-release-date-archive.zip` is in the current directory.

8. If necessary, start the Eclipse IDE and be sure you are in the Workbench.
9. Select "Install New Software" from the "Help" pull-down menu.
10. Click "Add".
11. Provide anything you want in the "Name" field.
12. Click "Archive" and browse to the ZIP file you built earlier. This ZIP file should not be "unzipped", and must be the `*archive.zip` file created by running the `build.sh` script.
13. Click the "OK" button.
14. Check the boxes that appear in the installation window to install the following:

```
Yocto Project SDK Plug-in
Yocto Project Documentation plug-in
```

15. Finish the installation by clicking through the appropriate buttons. You can click "OK" when prompted about installing software that contains unsigned content.
16. Restart the Eclipse IDE if necessary.

At this point you should be able to configure the Eclipse Yocto Plug-in as described in the "Configuring the Neon Eclipse Yocto Plug-in" section.

D.1.4. Configuring the Neon Eclipse Yocto Plug-In¶

Configuring the Neon Eclipse Yocto Plug-in involves setting the Cross Compiler options and the Target options. The configurations you choose become the default settings for all projects. You do have opportunities to change them later when you configure the project (see the following section).

To start, you need to do the following from within the Eclipse IDE:

1. Choose "Preferences" from the "Window" menu to display the Preferences Dialog.
2. Click "Yocto Project SDK" to display the configuration screen.

The following sub-sections describe how to configure the the plug-in.

> **Note**
> Throughout the descriptions, a start-to-finish example for preparing a QEMU image for use with Eclipse is referenced as the "wiki" and is linked to the example on the Cookbook guide to Making an Eclipse Debug Capable Image wiki page.

D.1.4.1. Configuring the Cross-Compiler Options¶

Cross Compiler options enable Eclipse to use your specific cross compiler toolchain. To configure these options, you must select the type of toolchain, point to the toolchain, specify the sysroot location, and select the target architecture.

- ***Selecting the Toolchain Type:*** Choose between "Standalone pre-built toolchain" and "Build system derived toolchain" for Cross Compiler Options.
  - ***Standalone Pre-built Toolchain:*** Select this type when you are using a stand-alone cross-toolchain. For example, suppose you are an application developer and do not need to build a target image. Instead, you just want to use an architecture-specific toolchain on an existing kernel and target root filesystem. In other words, you have downloaded and installed a pre-built toolchain for an existing image.
  - ***Build System Derived Toolchain:*** Select this type if you built the toolchain as part of the Build Directory. When you select "Build system derived toolchain", you are using the toolchain built and bundled inside the Build Directory. For example, suppose you created a suitable image using the steps in the wiki. In this situation, you would select "Build system derived toolchain".
- ***Specify the Toolchain Root Location:*** If you are using a stand-alone pre-built toolchain, you should be pointing to where it is installed (e.g. `/opt/poky/2.5`). See the "Installing the SDK" section for information about how the SDK is installed.

  If you are using a build system derived toolchain, the path you provide for the "Toolchain Root Location" field is the Build Directory from which you run the `bitbake` command (e.g `/home/scottrif/poky/build`).

  For more information, see the "Building an SDK Installer" section.
- ***Specify Sysroot Location:*** This location is where the root filesystem for the target hardware resides.

  This location depends on where you separately extracted and installed the target filesystem when you either built it or downloaded it.

> **Note**

If you downloaded the root filesystem for the target hardware rather than built it, you must download the `sato-sdk` image in order to build any c/c++ projects.

As an example, suppose you prepared an image using the steps in the wiki. If so, the `MY_QEMU_ROOTFS` directory is found in the Build Directory and you would browse to and select that directory (e.g. `/home/scottrif/build/MY_QEMU_ROOTFS`).

For more information on how to install the toolchain and on how to extract and install the sysroot filesystem, see the "Building an SDK Installer" section.

- ***Select the Target Architecture:*** The target architecture is the type of hardware you are going to use or emulate. Use the pull-down "Target Architecture" menu to make your selection. The pull-down menu should have the supported architectures. If the architecture you need is not listed in the menu, you will need to build the image. See the "Building a Simple Image" section of the Yocto Project Development Tasks Manual for more information. You can also see the wiki.

D.1.4.2. Configuring the Target Options¶

You can choose to emulate hardware using the QEMU emulator, or you can choose to run your image on actual hardware.

- ***QEMU:*** Select this option if you will be using the QEMU emulator. If you are using the emulator, you also need to locate the kernel and specify any custom options.

  If you selected the Build system derived toolchain, the target kernel you built will be located in the Build Directory in `tmp/deploy/images/machine` directory. As an example, suppose you performed the steps in the wiki. In this case, you specify your Build Directory path followed by the image (e.g. `/home/scottrif/poky/build/tmp/deploy/images/qemux86/bzImage-qemux86.bin`).

  If you selected the standalone pre-built toolchain, the pre-built image you downloaded is located in the directory you specified when you downloaded the image.

  Most custom options are for advanced QEMU users to further customize their QEMU instance. These options are specified between paired angled brackets. Some options must be specified outside the brackets. In particular, the options `serial`, `nographic`, and `kvm` must all be outside the brackets. Use the `man qemu` command to get help on all the options and their use. The following is an example:

  ```
  serial ‘<-m 256 -full-screen>’
  ```

  Regardless of the mode, Sysroot is already defined as part of the Cross-Compiler Options configuration in the "Sysroot Location:" field.

- ***External HW:*** Select this option if you will be using actual hardware.

Click the "Apply" and "OK" to save your plug-in configurations.

D.2. Creating the Project¶

You can create two types of projects: Autotools-based, or Makefile-based. This section describes how to create Autotools-based projects from within the Eclipse IDE. For information on creating Makefile-based projects in a terminal window, see the "Makefile-Based Projects" section.

> **Note**
> Do not use special characters in project names (e.g. spaces, underscores, etc.). Doing so can cause the configuration to fail.

To create a project based on a Yocto template and then display the source code, follow these steps:

1. Select "C Project" from the "File -> New" menu.
2. Expand "Yocto Project SDK Autotools Project".
3. Select "Hello World ANSI C Autotools Projects". This is an Autotools-based project based on a Yocto template.
4. Put a name in the "Project name:" field. Do not use hyphens as part of the name (e.g. "hello").
5. Click "Next".
6. Add appropriate information in the various fields.
7. Click "Finish".
8. If the "open perspective" prompt appears, click "Yes" so that you are in the C/C++ perspective.
9. The left-hand navigation pane shows your project. You can display your source by double clicking the project's source file.

D.3. Configuring the Cross-Toolchains¶

The earlier section, "Configuring the Neon Eclipse Yocto Plug-in", sets up the default project configurations. You can override these settings for a given project by following these steps:

1. Select "Yocto Project Settings" from the "Project -> Properties" menu. This selection brings up the Yocto Project Settings Dialog and allows you to make changes specific to an individual project.

   By default, the Cross Compiler Options and Target Options for a project are inherited from settings you provided using the Preferences Dialog as described earlier in the "Configuring the Neon Eclipse Yocto Plug-in" section. The Yocto Project Settings Dialog allows you to override those default settings for a given project.

2. Make or verify your configurations for the project and click "OK".

3. Right-click in the navigation pane and select "Reconfigure Project" from the pop-up menu. This selection reconfigures the project by running Autotools GNU utility programs such as Autoconf, Automake, and so forth in the workspace for your project. Click on the "Console" tab beneath your source code to see the results of reconfiguring your project.

D.4. Building the Project¶

To build the project select "Build All" from the "Project" menu. The console should update and you can note the cross-compiler you are using.

Note

When building "Yocto Project SDK Autotools" projects, the Eclipse IDE might display error messages for Functions/Symbols/Types that cannot be "resolved", even when the related include file is listed at the project navigator and when the project is able to build. For these cases only, it is recommended to add a new linked folder to the appropriate sysroot. Use these steps to add the linked folder:

1. Select the project.
2. Select "Folder" from the "File > New" menu.
3. In the "New Folder" Dialog, select "Link to alternate location (linked folder)".
4. Click "Browse" to navigate to the include folder inside the same sysroot location selected in the Yocto Project configuration preferences.
5. Click "OK".
6. Click "Finish" to save the linked folder.

D.5. Starting QEMU in User-Space NFS Mode¶

To start the QEMU emulator from within Eclipse, follow these steps:

Note

See the "Using the Quick EMUlator (QEMU)" chapter in the Yocto Project Development Tasks Manual for more information on using QEMU.

1. Expose and select "External Tools Configurations ..." from the "Run -> External Tools" menu.
2. Locate and select your image in the navigation panel to the left (e.g. `qemu_i586-poky-linux`).
3. Click "Run" to launch QEMU.

Note

The host on which you are running QEMU must have the `rpcbind` utility running to be able to make RPC calls on a server on that machine. If QEMU does not invoke and you receive error messages involving `rpcbind`, follow the suggestions to get the service running. As an example, on a new Ubuntu 16.04 LTS installation, you must do the following in order to get QEMU to launch:

```
$ sudo apt-get install rpcbind
```

After installing `rpcbind`, you need to edit the `/etc/init.d/rpcbind` file to include the following line:

```
OPTIONS="-i -w"
```

After modifying the file, you need to start the service:

```
$ sudo service portmap restart
```

4. If needed, enter your host root password in the shell window at the prompt. This sets up a `Tap 0` connection needed for running in user-space NFS mode.
5. Wait for QEMU to launch.
6. Once QEMU launches, you can begin operating within that environment. One useful task at this point would be to determine the IP Address for the user-space NFS by using the `ifconfig` command. The IP address of the QEMU machine appears in the xterm window. You can use this address to help you see which particular IP address the instance of QEMU is using.

D.6. Deploying and Debugging the Application¶

Once the QEMU emulator is running the image, you can deploy your application using the Eclipse IDE and then use the emulator to perform debugging. Follow these steps to deploy the application.

Note

Currently, Eclipse does not support SSH port forwarding. Consequently, if you need to run or debug a remote application using the host display, you must create a tunneling connection from outside Eclipse and keep that connection alive during your work. For example, in a new terminal, run the following:

```
$ ssh -XY user_name@remote_host_ip
```

Using the above form, here is an example:

```
$ ssh -XY root@192.168.7.2
```

After running the command, add the command to be executed in Eclipse's run configuration before the application as follows:

```
export DISPLAY=:10.0
```

Be sure to not destroy the connection during your QEMU session (i.e. do not exit out of or close that shell).

1. Select "Debug Configurations..." from the "Run" menu.
2. In the left area, expand "C/C++Remote Application".
3. Locate your project and select it to bring up a new tabbed view in the Debug Configurations Dialog.
4. Click on the "Debugger" tab to see the cross-tool debugger you are using. Be sure to change to the debugger perspective in Eclipse.
5. Click on the "Main" tab.
6. Create a new connection to the QEMU instance by clicking on "new".
7. Select "SSH", which means Secure Socket Shell. Optionally, you can select a TCF connection instead.
8. Click "Next".
9. Clear out the "Connection name" field and enter any name you want for the connection.
10. Put the IP address for the connection in the "Host" field. For QEMU, the default is "192.168.7.2". However, if a previous QEMU session did not exit cleanly, the IP address increments (e.g. "192.168.7.3").

Note

You can find the IP address for the current QEMU session by looking in the xterm that opens when you launch QEMU.

11. Enter "root", which is the default for QEMU, for the "User" field. Be sure to leave the password field empty.
12. Click "Finish" to close the New Connections Dialog.
13. If necessary, use the drop-down menu now in the "Connection" field and pick the IP Address you entered.
14. Assuming you are connecting as the root user, which is the default for QEMU x86-64 SDK images provided by the Yocto Project, in the "Remote Absolute File Path for C/C++ Application" field, browse to `/home/root/`*`ProjectName`* (e.g. `/home/root/hello`). You could also browse to any other path you have write access to on the target such as `/usr/bin`. This location is where your application will be located on the QEMU system. If you fail to browse to and specify an appropriate location, QEMU will not understand what to remotely launch. Eclipse is helpful in that it auto fills your application name for you assuming you browsed to a directory.

Tips

- If you are prompted to provide a username and to optionally set a password, be sure you provide "root" as the username and you leave the password field blank.
- If browsing to a directory fails or times out, but you can `ssh` into your QEMU or target from the command line and you have proxies set up, it is likely that Eclipse is sending the SSH traffic to a proxy. In this case, either use TCF , or click on "Configure proxy settings" in the connection dialog and add the target IP address to the "bypass proxy" section. You might also need to change "Active Provider" from Native to Manual.

15. Be sure you change to the "Debug" perspective in Eclipse.
16. Click "Debug"
17. Accept the debug perspective.

D.7. Using Linuxtools¶

As mentioned earlier in the manual, performance tools exist (Linuxtools) that enhance your development experience. These tools are aids in developing and debugging applications and images. You can run these tools from within the Eclipse IDE through the "Linuxtools" menu.

For information on how to configure and use these tools, see http://www.eclipse.org/linuxtools/.