OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Embedded Linux Systems with the Yocto Project™

Rudolf J. Streif

Embedded Linux Systems with the Yocto Project™

Listing 5-7 **showvars** Task

```
addtask showvars
do_showvars[nostamp] = "1"
python do_showvars() {
        # emit only the metadata that are variables and not functions
        isfunc = lambda key: bool(d.getVarFlag(key, 'func'))
        vars = sorted((key for key in bb.data.keys(d) \
               if not key.startswith('__')))
        for var in vars:
            if not isfunc(var):
                try:
                    val = d.getVar(var, True)
                except Exception as exc:
                    bb.plain('Expansion of %s threw %s: %s' % \
                             (var, exc.__class__.__name__, str(exc)))
                bb.plain('%s="%s"' % (var, val))
}
```

When used with

```
$ bitbake <target> -c showvars
```

the task lists all the variables but not the functions of the target's execution environment in alphabetical order.

5.4 Development Shell

Build failures that originate from compiling and linking of sources to objects, libraries, and executables are challenging to debug when cross-building. You cannot just change to the source directory, type `make`, examine the error messages, and correct the issue. Build environments for software packages are typically configured for native builds on the host system. A cross-build environment requires a different and often rather complicated setup for tools, header files, libraries, and more to correctly operate.

The Poky reference distribution creates cross-build environments for its own BitBake build process. Through the command `devshell`, it can also provide cross-build environments in a shell to the developer. The command

```
$ bitbake <target> -c devshell
```

launches a terminal with a cross-build environment for `target`. The setup of the cross-build environment exactly matches the one that Poky is using for its own builds. Within that shell you can use the development tools as you would for a native build on your build host. The environment references the correct cross-compilers as well as any header files, libraries, and other files required to build the software package.

If any of the dependencies of the software package you are targeting with the `devshell` command have not been built, Poky builds them beforehand.

If you are using BitBake from a graphical user interface with window manager, it automatically tries to determine the terminal program to use and open it in a new

window. The variable OE_TERMINAL controls what terminal program to use. It is typically set to auto. You can set it to one of the supported terminal programs in the conf/local.conf file of your build environment. The terminal program must be installed on your development host. You can disable the use of the devshell altogether by setting OE_TERMINAL = "none".

5.5 Dependency Graphs

In Chapter 4, "BitBake Build Engine," we saw how packages can declare direct build-time and runtime dependencies on other packages using the DEPENDS and RDEPENDS variables in their recipes. You can easily realize how this practice can lead to long and complex chains of dependencies.

BitBake, of course, must be able to resolve these dependency chains to build the packages in the correct order. Using its dependency resolver, BitBake can also create dependency graphs for analysis and debugging.

BitBake creates dependency graphs using the DOT plain text graph description language. DOT provides a simple way of describing undirected and directed graphs with nodes and edges in a language that can be read by humans and computer programs alike. Programs from the Graphviz[1] package as well as many others can read DOT files and render them into graphical representations.

To create dependency graphs for a target, invoke BitBake with the -g or --graphviz options. Using

```
$ bitbake -g <target>
```

creates the following dependency files for target:

- **pn-buildlist**: This file is not a DOT file but contains the list of packages in reverse build order starting with the target.
- **pn-depends.dot**: Contains the package dependencies in a directed graph declaring the nodes first and then the edges.
- **package-depends.dot**: Essentially the same as pn-depends.dot but declares the edges for a node right after the node. This file may be easier to read by humans because it groups the edges ending on a node with the node.
- **task-depends.dot**: Declares the dependencies on the task level.

BitBake also provides a built-in user interface for package dependencies, the dependency explorer. You can launch the dependency explorer with

```
$ bitbake -g -u depexp <target>
```

The dependency explorer lets you analyze build-time and runtime dependencies as well as reverse dependencies (the packages that depend on that package) in a graphical user interface, as shown in Figure 5-1.

1. www.graphviz.org

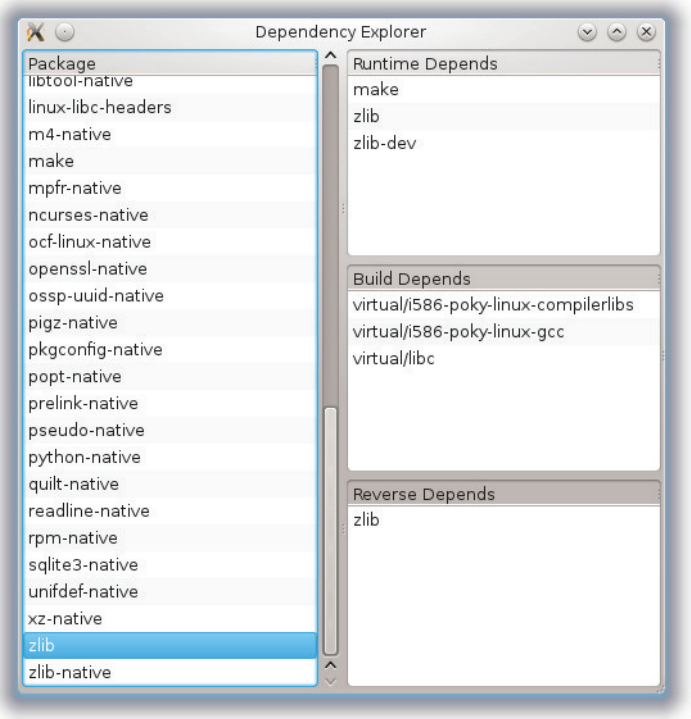

Figure 5-1 Dependency explorer

If the Graphviz package is installed on your development system, you can use it to create visual renditions of the dependency graphs:

```
$ dot -Tpng pn-depends.dot -o pn-depends.png
```

creates a PNG image from the DOT file.

5.6 Debugging Layers

BitBake's layer architecture provides an elegant way of organizing recipes. However, it also introduces complexity, particularly when multiple layers provide the same recipe and/or modify the same recipe with append files.

The bitbake-layers tool provides several functions that help with analyzing and debugging layers used by a build environment. Invoking

```
$ bitbake-layers help
```

provides a list of commands that can be used with the tool:

- **help**: Without any argument or by specifying `help` as argument, the tool shows a list of the available commands. If you provide a command, it shows additional help for that command.
- **show-layers**: Displays a list of the layers used by the build environment together with their path and priority.

```
$ bitbake-layers show-layers
layer                 path                                      priority
==========================================================================
meta                  /path/to/poky/meta                        5
meta-yocto            /path/to/poky/meta-yocto                  5
meta-yocto-bsp        /path/to/poky/meta-yocto-bsp              5
meta-mylayer          /path/to/meta-mylayer                     1
```

- **show-recipes**: Displays a list of recipes in alphabetical order including the layer providing it.

```
$ bitbake-layers show-recipes
Parsing recipes..done.
=== Available recipes: ===
acl:
  meta                 2.2.52
acpid:
  meta                 1.0.10
adt-installer:
  meta                 0.2.0
alsa-lib:
  meta                 1.0.27.2
alsa-state:
  meta                 0.2.0
alsa-tools:
  meta                 1.0.27
[...]
```

- **show-overlayed**: Displays a list of overlaid recipes. A recipe is overlaid if another recipe with the same name exists in a different layer.

```
$ bitbake-layers show-overlayed
Parsing recipes..done.
=== Overlayed recipes ===
mtd-utils:
  meta            1.5.0
  meta-mylayer    1.4.9
```

  When building an overlaid recipe, BitBake issues a warning and builds the recipe from the layer with the highest priority.

- **show-appends**: Displays a list of recipes with the files appending them. The appending files are shown in the order they are applied.

```
$ bitbake-layers show-appends
Parsing recipes..done.
=== Appended recipes ===
alsa-state.bb:
  /.../meta-yocto-bsp/recipes-bsp/alsa-state/alsa-state.bbappend
psplash_git.bb:
  /.../meta-yocto/recipes-core/psplash/psplash_git.bbappend
[...]
```

- **show-cross-depends**: Displays a list of all recipes that are dependent on metadata in other layers.

```
$ bitbake-layers show-cross-depends
Parsing recipes..done.
meta-yocto/recipes-core/tiny-init/tiny-init.bb RDEPENDS
           meta/recipes-core/busybox/busybox_1.22.1.bb
meta-yocto/recipes-core/tiny-init/tiny-init.bb inherits
           meta/classes/base.bbclass
meta-yocto/recipes-core/tiny-init/tiny-init.bb inherits
           meta/classes/patch.bbclass
meta-yocto/recipes-core/tiny-init/tiny-init.bb inherits
           meta/classes/terminal.bbclass
```

- **flatten <directory>**: Flattens the layer hierarchy by resolving recipe overlays and appends and writing the output into a single-layer directory provided by the parameter `directory`. Several rules apply:
  - If the layers contain overlaid recipes, the recipe from the layer with the highest priority is used. If the layers have the same priority, then the order of the layers in the `BBLAYERS` variable of the `conf/bblayers.conf` file of the build environment determine which recipe is used.
  - If multiple layers contain non-recipe files (such as images, patches, or similar) with the same name in the same subdirectory, they are overwritten by the one from the layer with the highest priority.
  - The `conf/layer.conf` file from the first layer listed in the `BBLAYERS` variable in `conf/bblayers.conf` of the build environment is used.
  - The contents of the append files are simply added to the respective recipe according to the layer priority or their order in the `BBLAYERS` variable if layers appending the same recipe have the same priority.

5.7 Summary

This chapter introduced a variety of tools provided by the OpenEmbedded build system to assist with troubleshooting build failures. Isolating the root cause of a problem is only the first step. The second step is finding a solution, which can be even more challenging. However, in many cases, other developers may have encountered the same or a similar problem, and searching the Internet frequently produces one or more discussions about the problem and potential solutions.

- Log files are a good starting point to identify the area of failure. Task log files contain the entire output of the commands a task executes.
- Inserting your own log messages into recipes and classes can help in pinpointing build failures.
- Executing specific tasks multiple times allows for comparing results. Task log files are not overwritten by consecutive runs.

- Printing metadata shows variables within their task contexts, including variable expansions and conditional assignments.
- The development shell allows execution of `make` targets within the same cross-build environment that BitBake uses.
- Dependency graphs support tracing build failures due to unresolved dependencies between software packages.
- The bitbake-layers utility provides a set of functions assisting with debugging build environments using multiple layers.