33 Debugging Tools and Techniques

The exact method for debugging build failures depends on the nature of the problem and on the system's area from which the bug originates. Standard debugging practices such as comparison against the last known working version with examination of the changes and the re-application of steps to identify the one causing the problem are valid for the Yocto Project just as they are for any other system. Even though it is impossible to detail every possible potential failure, this section provides some general tips to aid in debugging given a variety of situations.

Note

A useful feature for debugging is the error reporting tool. Configuring the Yocto Project to use this tool causes the OpenEmbedded build system to produce error reporting commands as part of the console output. You can enter the commands after the build completes to log error information into a common database, that can help you figure out what might be going wrong. For information on how to enable and use this feature, see the "Using the Error Reporting Tool" section.

The following list shows the debugging topics in the remainder of this section:

- "Viewing Logs from Failed Tasks" describes how to find and view logs from tasks that failed during the build process.
- "Viewing Variable Values" describes how to use the BitBake `-e` option to examine variable values after a recipe has been parsed.
- "Viewing Package Information with oe-pkgdata-util" describes how to use the `oe-pkgdata-util` utility to query PKGDATA_DIR and display package-related information for built packages.
- "Viewing Dependencies Between Recipes and Tasks" describes how to use the BitBake `-g` option to display recipe dependency information used during the build.
- "Viewing Task Variable Dependencies" describes how to use the `bitbake-dumpsig` command in conjunction with key subdirectories in the Build Directory to determine variable dependencies.

- “Running Specific Tasks” describes how to use several BitBake options (e.g. `-c`, `-C`, and `-f`) to run specific tasks in the build chain. It can be useful to run tasks “out-of-order” when trying isolate build issues.
- “General BitBake Problems” describes how to use BitBake’s `-D` debug output option to reveal more about what BitBake is doing during the build.
- “Building with No Dependencies” describes how to use the BitBake `-b` option to build a recipe while ignoring dependencies.
- “Recipe Logging Mechanisms” describes how to use the many recipe logging functions to produce debugging output and report errors and warnings.
- “Debugging Parallel Make Races” describes how to debug situations where the build consists of several parts that are run simultaneously and when the output or result of one part is not ready for use with a different part of the build that depends on that output.
- “Debugging With the GNU Project Debugger (GDB) Remotely” describes how to use GDB to allow you to examine running programs, which can help you fix problems.
- “Debugging with the GNU Project Debugger (GDB) on the Target” describes how to use GDB directly on target hardware for debugging.
- “Other Debugging Tips” describes miscellaneous debugging tips that can be useful.

33.1 Viewing Logs from Failed Tasks

You can find the log for a task in the file `${WORKDIR}/temp/log.do_`*taskname*. For example, the log for the do_compile task of the QEMU minimal image for the x86 machine (`qemux86`) might be in `tmp/work/qemux86-poky-linux/core-image-minimal/1.0-r0/temp/log.do_compile`. To see the commands BitBake ran to generate a log, look at the corresponding `run.do_`*taskname* file in the same directory.

`log.do_`*taskname* and `run.do_`*taskname* are actually symbolic links to `log.do_`*taskname*`.`*pid* and `log.run_`*taskname*`.`*pid*, where *pid* is the PID the task had when it ran. The symlinks always point to the files corresponding to the most recent run.

33.2 Viewing Variable Values

Sometimes you need to know the value of a variable as a result of BitBake’s parsing step. This could be because some unexpected behavior occurred in your project. Perhaps an attempt to modify a variable did not work out as expected.

BitBake's `-e` option is used to display variable values after parsing. The following command displays the variable values after the configuration files (i.e. `local.conf`, `bblayers.conf`, `bitbake.conf` and so forth) have been parsed:

```
$ bitbake -e
```

The following command displays variable values after a specific recipe has been parsed. The variables include those from the configuration as well:

```
$ bitbake -e recipename
```

Note

Each recipe has its own private set of variables (datastore). Internally, after parsing the configuration, a copy of the resulting datastore is made prior to parsing each recipe. This copying implies that variables set in one recipe will not be visible to other recipes.

Likewise, each task within a recipe gets a private datastore based on the recipe datastore, which means that variables set within one task will not be visible to other tasks.

In the output of `bitbake -e`, each variable is preceded by a description of how the variable got its value, including temporary values that were later overridden. This description also includes variable flags (varflags) set on the variable. The output can be very helpful during debugging.

Variables that are exported to the environment are preceded by `export` in the output of `bitbake -e`. See the following example:

```
export CC="i586-poky-linux-gcc -m32 -march=i586 --
sysroot=/home/ulf/poky/build/tmp/sysroots/qemux86"
```

In addition to variable values, the output of the `bitbake -e` and `bitbake -e` recipe commands includes the following information:

- The output starts with a tree listing all configuration files and classes included globally, recursively listing the files they include or inherit in turn. Much of the behavior of the OpenEmbedded build system (including the behavior of the

Normal Recipe Build Tasks) is implemented in the base class and the classes it inherits, rather than being built into BitBake itself.
- After the variable values, all functions appear in the output. For shell functions, variables referenced within the function body are expanded. If a function has been modified using overrides or using override-style operators like `:append` and `:prepend`, then the final assembled function body appears in the output.

33.3 Viewing Package Information with `oe-pkgdata-util`

You can use the `oe-pkgdata-util` command-line utility to query PKGDATA_DIR and display various package-related information. When you use the utility, you must use it to view information on packages that have already been built.

Following are a few of the available `oe-pkgdata-util` subcommands.

> **Note**
>
> You can use the standard * and ? globbing wildcards as part of package names and paths.

- `oe-pkgdata-util list-pkgs [pattern]`: Lists all packages that have been built, optionally limiting the match to packages that match pattern.
- `oe-pkgdata-util list-pkg-files package ...`: Lists the files and directories contained in the given packages.

  > **Note**
  >
  > A different way to view the contents of a package is to look at the `${WORKDIR}/packages-split` directory of the recipe that generates the package. This directory is created by the do_package task and has one subdirectory for each package the recipe generates, which contains the files stored in that package.
  >
  > If you want to inspect the `${WORKDIR}/packages-split` directory, make sure that rm_work is not enabled when you build the recipe.

- `oe-pkgdata-util find-path path ...`: Lists the names of the packages that contain the given paths. For example, the following tells us that `/usr/share/man/man1/make.1` is contained in the `make-doc` package:

```
$ oe-pkgdata-util find-path /usr/share/man/man1/make.1
make-doc: /usr/share/man/man1/make.1
```

- `oe-pkgdata-util lookup-recipe package ...`: Lists the name of the recipes that produce the given packages.

For more information on the `oe-pkgdata-util` command, use the help facility:

```
$ oe-pkgdata-util --help
$ oe-pkgdata-util subcommand --help
```

33.4 Viewing Dependencies Between Recipes and Tasks

Sometimes it can be hard to see why BitBake wants to build other recipes before the one you have specified. Dependency information can help you understand why a recipe is built.

To generate dependency information for a recipe, run the following command:

```
$ bitbake -g recipename
```

This command writes the following files in the current directory:

- `pn-buildlist`: A list of recipes/targets involved in building *recipename*. "Involved" here means that at least one task from the recipe needs to run when building *recipename* from scratch. Targets that are in ASSUME_PROVIDED are not listed.
- `task-depends.dot`: A graph showing dependencies between tasks.

The graphs are in DOT format and can be converted to images (e.g. using the `dot` tool from Graphviz).

Note

- DOT files use a plain text format. The graphs generated using the `bitbake -g` command are often so large as to be difficult to read without special pruning (e.g. with BitBake's `-I` option) and processing. Despite the form and size of the graphs, the corresponding `.dot` files can still be possible to read and provide useful information.

As an example, the `task-depends.dot` file contains lines such as the following:

```
"libxslt.do_configure" -> "libxml2.do_populate_sysroot"
```

The above example line reveals that the do_configure task in `libxslt` depends on the do_populate_sysroot task in `libxml2`, which is a normal DEPENDS dependency between the two recipes.

- For an example of how `.dot` files can be processed, see the `scripts/contrib/graph-tool` Python script, which finds and displays paths between graph nodes.

You can use a different method to view dependency information by using the following command:

```
$ bitbake -g -u taskexp recipename
```

This command displays a GUI window from which you can view build-time and runtime dependencies for the recipes involved in building recipename.

33.5 Viewing Task Variable Dependencies

As mentioned in the "Checksums (Signatures)" section of the BitBake User Manual, BitBake tries to automatically determine what variables a task depends on so that it can rerun the task if any values of the variables change. This determination is usually reliable. However, if you do things like construct variable names at runtime, then you might have to manually declare dependencies on those variables using `vardeps` as described in the "Variable Flags" section of the BitBake User Manual.

If you are unsure whether a variable dependency is being picked up automatically for a given task, you can list the variable dependencies BitBake has determined by doing the following:

1. Build the recipe containing the task:

```
$ bitbake recipename
```

2. Inside the STAMPS_DIR directory, find the signature data (`sigdata`) file that corresponds to the task. The `sigdata` files contain a pickled Python database of all the metadata that went into creating the input checksum for the task. As an example, for the do_fetch task of the `db` recipe, the `sigdata` file might be found in the following location:

```
${BUILDDIR}/tmp/stamps/i586-poky-linux/db/6.0.30-
r1.do_fetch.sigdata.7c048c18222b16ff0bcee2000ef648b1
```

   For tasks that are accelerated through the shared state (sstate) cache, an additional `siginfo` file is written into SSTATE_DIR along with the cached task output. The `siginfo` files contain exactly the same information as `sigdata` files.

3. Run `bitbake-dumpsig` on the `sigdata` or `siginfo` file. Here is an example:

```
$ bitbake-dumpsig ${BUILDDIR}/tmp/stamps/i586-poky-linux/db/6.0.30-
r1.do_fetch.sigdata.7c048c18222b16ff0bcee2000ef648b1
```

   In the output of the above command, you will find a line like the following, which lists all the (inferred) variable dependencies for the task. This list also includes indirect dependencies from variables depending on other variables, recursively:

```
Task dependencies: ['PV', 'SRCREV', 'SRC_URI', 'SRC_URI[md5sum]',
'SRC_URI[sha256sum]', 'base_do_fetch']
```

   Note

   Functions (e.g. `base_do_fetch`) also count as variable dependencies. These functions in turn depend on the variables they reference.

   The output of `bitbake-dumpsig` also includes the value each variable had, a list of dependencies for each variable, and BB_BASEHASH_IGNORE_VARS information.

There is also a `bitbake-diffsigs` command for comparing two `siginfo` or `sigdata` files. This command can be helpful when trying to figure out what changed between two versions of a task. If you call `bitbake-diffsigs` with just one file, the command behaves like `bitbake-dumpsig`.

You can also use BitBake to dump out the signature construction information without executing tasks by using either of the following BitBake command-line options:

```
--dump-signatures=SIGNATURE_HANDLER
-S SIGNATURE_HANDLER
```

Note

Two common values for *SIGNATURE_HANDLER* are “none” and “printdiff”, which dump only the signature or compare the dumped signature with the cached one, respectively.

Using BitBake with either of these options causes BitBake to dump out `sigdata` files in the `stamps` directory for every task it would have executed instead of building the specified target package.

33.6 Viewing Metadata Used to Create the Input Signature of a Shared State Task

Seeing what metadata went into creating the input signature of a shared state (sstate) task can be a useful debugging aid. This information is available in signature information (`siginfo`) files in SSTATE_DIR. For information on how to view and interpret information in `siginfo` files, see the “Viewing Task Variable Dependencies” section.

For conceptual information on shared state, see the “Shared State” section in the Yocto Project Overview and Concepts Manual.

33.7 Invalidating Shared State to Force a Task to Run

The OpenEmbedded build system uses checksums and Shared State cache to avoid unnecessarily rebuilding tasks. Collectively, this scheme is known as “shared state code”.

As with all schemes, this one has some drawbacks. It is possible that you could make implicit changes to your code that the checksum calculations do not take into account. These implicit changes affect a task’s output but do not trigger the shared state code into rebuilding a recipe. Consider an example during which a

tool changes its output. Assume that the output of `rpmdeps` changes. The result of the change should be that all the `package` and `package_write_rpm` shared state cache items become invalid. However, because the change to the output is external to the code and therefore implicit, the associated shared state cache items do not become invalidated. In this case, the build process uses the cached items rather than running the task again. Obviously, these types of implicit changes can cause problems.

To avoid these problems during the build, you need to understand the effects of any changes you make. Realize that changes you make directly to a function are automatically factored into the checksum calculation. Thus, these explicit changes invalidate the associated area of shared state cache. However, you need to be aware of any implicit changes that are not obvious changes to the code and could affect the output of a given task.

When you identify an implicit change, you can easily take steps to invalidate the cache and force the tasks to run. The steps you can take are as simple as changing a function's comments in the source code. For example, to invalidate package shared state files, change the comment statements of do_package or the comments of one of the functions it calls. Even though the change is purely cosmetic, it causes the checksum to be recalculated and forces the build system to run the task again.

Note

For an example of a commit that makes a cosmetic change to invalidate shared state, see this commit.

33.8 Running Specific Tasks

Any given recipe consists of a set of tasks. The standard BitBake behavior in most cases is: do_fetch, do_unpack, do_patch, do_configure, do_compile, do_install, do_package, do_package_write_*, and do_build. The default task is do_build and any tasks on which it depends build first. Some tasks, such as do_devshell, are not part of the default build chain. If you wish to run a task that is not part of the default build chain, you can use the `-c` option in BitBake. Here is an example:

```
$ bitbake matchbox-desktop -c devshell
```

The `-c` option respects task dependencies, which means that all other tasks (including tasks from other recipes) that the specified task depends on will be run before the task. Even when you manually specify a task to run with `-c`, BitBake will only run the task if it considers it "out of date". See the "Stamp Files and the Rerunning of Tasks" section in the Yocto Project Overview and Concepts Manual for how BitBake determines whether a task is "out of date".

If you want to force an up-to-date task to be rerun (e.g. because you made manual modifications to the recipe's WORKDIR that you want to try out), then you can use the `-f` option.

Note

The reason `-f` is never required when running the do_devshell task is because the [nostamp] variable flag is already set for the task.

The following example shows one way you can use the `-f` option:

```
$ bitbake matchbox-desktop
          .
          .
make some changes to the source code in the work directory
          .
          .
$ bitbake matchbox-desktop -c compile -f
$ bitbake matchbox-desktop
```

This sequence first builds and then recompiles `matchbox-desktop`. The last command reruns all tasks (basically the packaging tasks) after the compile. BitBake recognizes that the do_compile task was rerun and therefore understands that the other tasks also need to be run again.

Another, shorter way to rerun a task and all Normal Recipe Build Tasks that depend on it is to use the `-C` option.

Note

This option is upper-cased and is separate from the `-c` option, which is lower-cased.

Using this option invalidates the given task and then runs the do_build task, which is the default task if no task is given, and the tasks on which it depends. You could replace the final two commands in the previous example with the following single command:

```
$ bitbake matchbox-desktop -C compile
```

Internally, the `-f` and `-C` options work by tainting (modifying) the input checksum of the specified task. This tainting indirectly causes the task and its dependent tasks to be rerun through the normal task dependency mechanisms.

Note

BitBake explicitly keeps track of which tasks have been tainted in this fashion, and will print warnings such as the following for builds involving such tasks:

```
WARNING: /home/ulf/poky/meta/recipes-sato/matchbox-desktop/matchbox-
desktop_2.1.bb.do_compile is tainted from a forced run
```

The purpose of the warning is to let you know that the work directory and build output might not be in the clean state they would be in for a “normal” build, depending on what actions you took. To get rid of such warnings, you can remove the work directory and rebuild the recipe, as follows:

```
$ bitbake matchbox-desktop -c clean
$ bitbake matchbox-desktop
```

You can view a list of tasks in a given package by running the do_listtasks task as follows:

```
$ bitbake matchbox-desktop -c listtasks
```

The results appear as output to the console and are also in the file `${WORKDIR}/temp/log.do_listtasks`.

33.9 General BitBake Problems

You can see debug output from BitBake by using the `-D` option. The debug output gives more information about what BitBake is doing and the reason behind it. Each `-D` option you use increases the logging level. The most common usage is `-DDD`.

The output from `bitbake -DDD -v targetname` can reveal why BitBake chose a certain version of a package or why BitBake picked a certain provider. This command could also help you in a situation where you think BitBake did something unexpected.

33.10 Building with No Dependencies

To build a specific recipe (`.bb` file), you can use the following command form:

```
$ bitbake -b somepath/somerecipe.bb
```

This command form does not check for dependencies. Consequently, you should use it only when you know existing dependencies have been met.

> **Note**
>
> You can also specify fragments of the filename. In this case, BitBake checks for a unique match.

33.11 Recipe Logging Mechanisms

The Yocto Project provides several logging functions for producing debugging output and reporting errors and warnings. For Python functions, the following logging functions are available. All of these functions log to `${T}/log.do_`*task*, and can also log to standard output (stdout) with the right settings:

- `bb.plain(msg)`: Writes msg as is to the log while also logging to stdout.
- `bb.note(msg)`: Writes “NOTE: msg” to the log. Also logs to stdout if BitBake is called with “-v”.
- `bb.debug(level, msg)`: Writes “DEBUG: msg” to the log. Also logs to stdout if the log level is greater than or equal to level. See the “Usage and syntax” option in the BitBake User Manual for more information.
- `bb.warn(msg)`: Writes “WARNING: msg” to the log while also logging to stdout.
- `bb.error(msg)`: Writes “ERROR: msg” to the log while also logging to standard out (stdout).

> **Note**

Calling this function does not cause the task to fail.

- `bb.fatal(msg)`: This logging function is similar to `bb.error(msg)` but also causes the calling task to fail.

  Note

  `bb.fatal()` raises an exception, which means you do not need to put a "return" statement after the function.

The same logging functions are also available in shell functions, under the names `bbplain`, `bbnote`, `bbdebug`, `bbwarn`, `bberror`, and `bbfatal`. The logging class implements these functions. See that class in the `meta/classes` folder of the Source Directory for information.

33.11.1 Logging With Python

When creating recipes using Python and inserting code that handles build logs, keep in mind the goal is to have informative logs while keeping the console as "silent" as possible. Also, if you want status messages in the log, use the "debug" loglevel.

Following is an example written in Python. The code handles logging for a function that determines the number of tasks needed to be run. See the "do_listtasks" section for additional information:

```
python do_listtasks() {
    bb.debug(2, "Starting to figure out the task list")
    if noteworthy_condition:
        bb.note("There are 47 tasks to run")
    bb.debug(2, "Got to point xyz")
    if warning_trigger:
        bb.warn("Detected warning_trigger, this might be a problem later.")
    if recoverable_error:
        bb.error("Hit recoverable_error, you really need to fix this!")
    if fatal_error:
        bb.fatal("fatal_error detected, unable to print the task list")
    bb.plain("The tasks present are abc")
    bb.debug(2, "Finished figuring out the tasklist")
}
```

33.11.2 Logging With Bash

When creating recipes using Bash and inserting code that handles build logs, you have the same goals — informative with minimal console output. The syntax you use for recipes written in Bash is similar to that of recipes written in Python

described in the previous section.

Following is an example written in Bash. The code logs the progress of the `do_my_function` function:

```
do_my_function() {
    bbdebug 2 "Running do_my_function"
    if [ exceptional_condition ]; then
        bbnote "Hit exceptional_condition"
    fi
    bbdebug 2  "Got to point xyz"
    if [ warning_trigger ]; then
        bbwarn "Detected warning_trigger, this might cause a problem later."
    fi
    if [ recoverable_error ]; then
        bberror "Hit recoverable_error, correcting"
    fi
    if [ fatal_error ]; then
        bbfatal "fatal_error detected"
    fi
    bbdebug 2 "Completed do_my_function"
}
```

33.12 Debugging Parallel Make Races

A parallel `make` race occurs when the build consists of several parts that are run simultaneously and a situation occurs when the output or result of one part is not ready for use with a different part of the build that depends on that output. Parallel make races are annoying and can sometimes be difficult to reproduce and fix. However, there are some simple tips and tricks that can help you debug and fix them. This section presents a real-world example of an error encountered on the Yocto Project autobuilder and the process used to fix it.

Note

If you cannot properly fix a `make` race condition, you can work around it by clearing either the PARALLEL_MAKE or PARALLEL_MAKEINST variables.

33.12.1 The Failure

For this example, assume that you are building an image that depends on the “neard” package. And, during the build, BitBake runs into problems and creates the following output.

Note

This example log file has longer lines artificially broken to make the listing easier to read.

If you examine the output or the log file, you see the failure during `make`:

```
| DEBUG: SITE files ['endian-little', 'bit-32', 'ix86-common', 'common-linux', 'common-
glibc', 'i586-linux', 'common']
| DEBUG: Executing shell function do_compile
| NOTE: make -j 16
| make --no-print-directory all-am
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/types.h include/near/types.h
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/log.h include/near/log.h
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/plugin.h include/near/plugin.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/tag.h include/near/tag.h
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/adapter.h include/near/adapter.h
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/ndef.h include/near/ndef.h
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/tlv.h include/near/tlv.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/setting.h include/near/setting.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/device.h include/near/device.h
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/nfc_copy.h include/near/nfc_copy.h
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/snep.h include/near/snep.h
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/version.h include/near/version.h
| ln -s /home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/work/i586-poky-
linux/neard/
  0.14-r0/neard-0.14/include/dbus.h include/near/dbus.h
| ./src/genbuiltin nfctype1 nfctype2 nfctype3 nfctype4 p2p > src/builtin.h
| i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/home/pokybuild/yocto-
autobuilder/nightly-x86/
  build/build/tmp/sysroots/qemux86 -DHAVE_CONFIG_H -I. -I./include -I./src -I./gdbus  -
I/home/pokybuild/
  yocto-autobuilder/nightly-x86/build/build/tmp/sysroots/qemux86/usr/include/glib-2.0
```

```
  -I/home/pokybuild/yocto-autobuilder/nightly-x86/build/build/tmp/sysroots/qemux86/usr/
  lib/glib-2.0/include  -I/home/pokybuild/yocto-autobuilder/nightly-x86/build/build/
  tmp/sysroots/qemux86/usr/include/dbus-1.0 -I/home/pokybuild/yocto-autobuilder/
  nightly-x86/build/build/tmp/sysroots/qemux86/usr/lib/dbus-1.0/include  -
I/home/pokybuild/yocto-autobuilder/
  nightly-x86/build/build/tmp/sysroots/qemux86/usr/include/libnl3
  -DNEAR_PLUGIN_BUILTIN -DPLUGINDIR=\""/usr/lib/near/plugins"\"
  -DCONFIGDIR=\""/etc/neard\"" -O2 -pipe -g -feliminate-unused-debug-types -c
  -o tools/snep-send.o tools/snep-send.c
| In file included from tools/snep-send.c:16:0:
| tools/../src/near.h:41:23: fatal error: near/dbus.h: No such file or directory
|  #include <near/dbus.h>
|                       ^
| compilation terminated.
| make[1]: *** [tools/snep-send.o] Error 1
| make[1]: *** Waiting for unfinished jobs....
| make: *** [all] Error 2
| ERROR: oe_runmake failed
```

33.12.2 Reproducing the Error

Because race conditions are intermittent, they do not manifest themselves every time you do the build. In fact, most times the build will complete without problems even though the potential race condition exists. Thus, once the error surfaces, you need a way to reproduce it.

In this example, compiling the “neard” package is causing the problem. So the first thing to do is build “neard” locally. Before you start the build, set the PARALLEL_MAKE variable in your `local.conf` file to a high number (e.g. “-j 20”). Using a high value for PARALLEL_MAKE increases the chances of the race condition showing up:

```
$ bitbake neard
```

Once the local build for “neard” completes, start a `devshell` build:

```
$ bitbake neard -c devshell
```

For information on how to use a `devshell`, see the “Using a Development Shell” section.

In the `devshell`, do the following:

```
$ make clean
$ make tools/snep-send.o
```

The `devshell` commands cause the failure to clearly be visible. In this case, there is a missing dependency for the `neard` Makefile target. Here is some abbreviated, sample output with the missing dependency clearly visible at the end:

```
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/home/scott-lenovo/......
   .
   .
   .
tools/snep-send.c
In file included from tools/snep-send.c:16:0:
tools/../src/near.h:41:23: fatal error: near/dbus.h: No such file or directory
 #include <near/dbus.h>
                   ^
compilation terminated.
make: *** [tools/snep-send.o] Error 1
$
```

33.12.3 Creating a Patch for the Fix

Because there is a missing dependency for the Makefile target, you need to patch the `Makefile.am` file, which is generated from `Makefile.in`. You can use Quilt to create the patch:

```
$ quilt new parallelmake.patch
Patch patches/parallelmake.patch is now on top
$ quilt add Makefile.am
File Makefile.am added to patch patches/parallelmake.patch
```

For more information on using Quilt, see the "Using Quilt in Your Workflow" section.

At this point you need to make the edits to `Makefile.am` to add the missing dependency. For our example, you have to add the following line to the file:

```
tools/snep-send.$(OBJEXT): include/near/dbus.h
```

Once you have edited the file, use the `refresh` command to create the patch:

```
$ quilt refresh
Refreshed patch patches/parallelmake.patch
```

Once the patch file is created, you need to add it back to the originating recipe folder. Here is an example assuming a top-level Source Directory named `poky`:

```
$ cp patches/parallelmake.patch poky/meta/recipes-connectivity/neard/neard
```

The final thing you need to do to implement the fix in the build is to update the "neard" recipe (i.e. `neard-0.14.bb`) so that the SRC_URI statement includes the patch file. The recipe file is in the folder above the patch. Here is what the edited SRC_URI statement would look like:

```
SRC_URI = "${KERNELORG_MIRROR}/linux/network/nfc/${BPN}-${PV}.tar.xz \
           file://neard.in \
           file://neard.service.in \
           file://parallelmake.patch \
          "
```

With the patch complete and moved to the correct folder and the SRC_URI statement updated, you can exit the `devshell`:

```
$ exit
```

33.12.4 Testing the Build

With everything in place, you can get back to trying the build again locally:

```
$ bitbake neard
```

This build should succeed.

Now you can open up a `devshell` again and repeat the clean and make operations as follows:

```
$ bitbake neard -c devshell
$ make clean
$ make tools/snep-send.o
```

The build should work without issue.

As with all solved problems, if they originated upstream, you need to submit the fix for the recipe in OE-Core and upstream so that the problem is taken care of at its source. See the “Contributing Changes to a Component” section for more information.

33.13 Debugging With the GNU Project Debugger (GDB) Remotely

GDB allows you to examine running programs, which in turn helps you to understand and fix problems. It also allows you to perform post-mortem style analysis of program crashes. GDB is available as a package within the Yocto Project and is installed in SDK images by default. See the “Images” chapter in the Yocto Project Reference Manual for a description of these images. You can find information on GDB at https://sourceware.org/gdb/.

> **Note**
>
> For best results, install debug (`-dbg`) packages for the applications you are going to debug. Doing so makes extra debug symbols available that give you more meaningful output.

Sometimes, due to memory or disk space constraints, it is not possible to use GDB directly on the remote target to debug applications. These constraints arise because GDB needs to load the debugging information and the binaries of the process being debugged. Additionally, GDB needs to perform many computations to locate information such as function names, variable names and values, stack traces and so forth — even before starting the debugging process. These extra computations place more load on the target system and can alter the characteristics of the program being debugged.

To help get past the previously mentioned constraints, there are two methods you can use: running a debuginfod server and using gdbserver.

33.13.1 Using the debuginfod server method

`debuginfod` from `elfutils` is a way to distribute `debuginfo` files. Running a `debuginfod` server makes debug symbols readily available, which means you don't need to download debugging information and the binaries of the process being debugged. You can just fetch debug symbols from the server.

To run a `debuginfod` server, you need to do the following:

- Ensure that `debuginfod` is present in DISTRO_FEATURES (it already is in `OpenEmbedded-core` defaults and `poky` reference distribution). If not, set in your distro config file or in `local.conf`:

  ```
  DISTRO_FEATURES:append = " debuginfod"
  ```

  This distro feature enables the server and client library in `elfutils`, and enables `debuginfod` support in clients (at the moment, `gdb` and `binutils`).

- Run the following commands to launch the `debuginfod` server on the host:

  ```
  $ oe-debuginfod
  ```

- To use `debuginfod` on the target, you need to know the ip:port where `debuginfod` is listening on the host (port defaults to 8002), and export that into the shell environment, for example in `qemu`:

  ```
  root@qemux86-64:~# export DEBUGINFOD_URLS="http://192.168.7.1:8002/"
  ```

- Then debug info fetching should simply work when running the target `gdb`, `readelf` or `objdump`, for example:

  ```
  root@qemux86-64:~# gdb /bin/cat
  ...
  Reading symbols from /bin/cat...
  Downloading separate debug info for /bin/cat...
  Reading symbols from
  /home/root/.cache/debuginfod_client/923dc4780cfbc545850c616bffa884b6b5eaf322/debuginfo
  ```

- It’s also possible to use `debuginfod-find` to just query the server:

```
root@qemux86-64:~# debuginfod-find debuginfo /bin/ls
/home/root/.cache/debuginfod_client/356edc585f7f82d46f94fcb87a86a3fe2d2e60bd/debuginfo
```

33.13.2 Using the gdbserver method

gdbserver, which runs on the remote target and does not load any debugging information from the debugged process. Instead, a GDB instance processes the debugging information that is run on a remote computer - the host GDB. The host GDB then sends control commands to gdbserver to make it stop or start the debugged program, as well as read or write memory regions of that debugged program. All the debugging information loaded and processed as well as all the heavy debugging is done by the host GDB. Offloading these processes gives the gdbserver running on the target a chance to remain small and fast.

Because the host GDB is responsible for loading the debugging information and for doing the necessary processing to make actual debugging happen, you have to make sure the host can access the unstripped binaries complete with their debugging information and also be sure the target is compiled with no optimizations. The host GDB must also have local access to all the libraries used by the debugged program. Because gdbserver does not need any local debugging information, the binaries on the remote target can remain stripped. However, the binaries must also be compiled without optimization so they match the host’s binaries.

To remain consistent with GDB documentation and terminology, the binary being debugged on the remote target machine is referred to as the “inferior” binary. For documentation on GDB see the GDB site.

The following steps show you how to debug using the GNU project debugger.

1. ***Configure your build system to construct the companion debug filesystem:***

   In your `local.conf` file, set the following:

```
IMAGE_GEN_DEBUGFS = "1"
IMAGE_FSTYPES_DEBUGFS = "tar.bz2"
```

These options cause the OpenEmbedded build system to generate a special companion filesystem fragment, which contains the matching source and debug symbols to your deployable filesystem. The build system does this by looking at what is in the deployed filesystem, and pulling the corresponding `-dbg` packages.

The companion debug filesystem is not a complete filesystem, but only contains the debug fragments. This filesystem must be combined with the full filesystem for debugging. Subsequent steps in this procedure show how to combine the partial filesystem with the full filesystem.

2. ***Configure the system to include gdbserver in the target filesystem:***

   Make the following addition in your `local.conf` file:

   ```
   EXTRA_IMAGE_FEATURES:append = " tools-debug"
   ```

   The change makes sure the `gdbserver` package is included.

3. ***Build the environment:***

   Use the following command to construct the image and the companion Debug Filesystem:

   ```
   $ bitbake image
   ```

   Build the cross GDB component and make it available for debugging. Build the SDK that matches the image. Building the SDK is best for a production build that can be used later for debugging, especially during long term maintenance:

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

   Alternatively, you can build the minimal toolchain components that match the target. Doing so creates a smaller than typical SDK and only contains a minimal set of components with which to build simple test applications, as well as run the debugger:

   ```
   $ bitbake meta-toolchain
   ```

   A final method is to build Gdb itself within the build system:

```
$ bitbake gdb-cross-<architecture>
```

Doing so produces a temporary copy of `cross-gdb` you can use for debugging during development. While this is the quickest approach, the two previous methods in this step are better when considering long-term maintenance strategies.

Note

If you run `bitbake gdb-cross`, the OpenEmbedded build system suggests the actual image (e.g. `gdb-cross-i586`). The suggestion is usually the actual name you want to use.

4. ***Set up the `debugfs`:***

   Run the following commands to set up the `debugfs`:

   ```
   $ mkdir debugfs
   $ cd debugfs
   $ tar xvfj build-dir/tmp/deploy/images/machine/image.rootfs.tar.bz2
   $ tar xvfj build-dir/tmp/deploy/images/machine/image-dbg.rootfs.tar.bz2
   ```

5. ***Set up GDB:***

   Install the SDK (if you built one) and then source the correct environment file. Sourcing the environment file puts the SDK in your `PATH` environment variable and sets `$GDB` to the SDK's debugger.

   If you are using the build system, Gdb is located in *build-dir*`/tmp/sysroots/`*host*`/usr/bin/`*architecture*`/`*architecture*`-gdb`

6. ***Boot the target:***

   For information on how to run QEMU, see the QEMU Documentation.

   Note

   Be sure to verify that your host can access the target via TCP.

7. ***Debug a program:***

   Debugging a program involves running gdbserver on the target and then running Gdb on the host. The example in this step debugs `gzip`:

   ```
   root@qemux86:~# gdbserver localhost:1234 /bin/gzip –help
   ```

For additional gdbserver options, see the GDB Server Documentation.

After running gdbserver on the target, you need to run Gdb on the host and configure it and connect to the target. Use these commands:

```
$ cd directory-holding-the-debugfs-directory
$ arch-gdb
(gdb) set sysroot debugfs
(gdb) set substitute-path /usr/src/debug debugfs/usr/src/debug
(gdb) target remote IP-of-target:1234
```

At this point, everything should automatically load (i.e. matching binaries, symbols and headers).

Note

The Gdb `set` commands in the previous example can be placed into the users `~/.gdbinit` file. Upon starting, Gdb automatically runs whatever commands are in that file.

8. ***Deploying without a full image rebuild:***

   In many cases, during development you want a quick method to deploy a new binary to the target and debug it, without waiting for a full image build.

   One approach to solving this situation is to just build the component you want to debug. Once you have built the component, copy the executable directly to both the target and the host `debugfs`.

   If the binary is processed through the debug splitting in OpenEmbedded, you should also copy the debug items (i.e. `.debug` contents and corresponding `/usr/src/debug` files) from the work directory. Here is an example:

```
$ bitbake bash
$ bitbake -c devshell bash
$ cd ..
$ scp packages-split/bash/bin/bash target:/bin/bash
$ cp -a packages-split/bash-dbg/\* path/debugfs
```

33.14 Debugging with the GNU Project Debugger (GDB) on the Target

The previous section addressed using GDB remotely for debugging purposes, which is the most usual case due to the inherent hardware limitations on many embedded devices. However, debugging in the target hardware itself is also

possible with more powerful devices. This section describes what you need to do in order to support using GDB to debug on the target hardware.

To support this kind of debugging, you need do the following:

- Ensure that GDB is on the target. You can do this by making the following addition to your `local.conf` file:

```
EXTRA_IMAGE_FEATURES:append = " tools-debug"
```

- Ensure that debug symbols are present. You can do so by adding the corresponding `-dbg` package to IMAGE_INSTALL:

```
IMAGE_INSTALL:append = " packagename-dbg"
```

  Alternatively, you can add the following to `local.conf` to include all the debug symbols:

```
EXTRA_IMAGE_FEATURES:append = " dbg-pkgs"
```

Note

To improve the debug information accuracy, you can reduce the level of optimization used by the compiler. For example, when adding the following line to your `local.conf` file, you will reduce optimization from FULL_OPTIMIZATION of “-O2” to DEBUG_OPTIMIZATION of “-O -fno-omit-frame-pointer”:

```
DEBUG_BUILD = "1"
```

Consider that this will reduce the application’s performance and is recommended only for debugging purposes.

33.15 Other Debugging Tips

Here are some other tips that you might find useful:

- When adding new packages, it is worth watching for undesirable items making their way into compiler command lines. For example, you do not want references to local system files like `/usr/lib/` or `/usr/include/`.
- If you want to remove the `psplash` boot splashscreen, add `psplash=false` to the kernel command line. Doing so prevents `psplash` from loading and thus allows you to see the console. It is also possible to switch out of the splashscreen by switching the virtual console (e.g. Fn+Left or Fn+Right on a Zaurus).
- Removing TMPDIR (usually `tmp/`, within the Build Directory) can often fix temporary build issues. Removing TMPDIR is usually a relatively cheap operation, because task output will be cached in SSTATE_DIR (usually `sstate-cache/`, which is also in the Build Directory).

  Note

  Removing TMPDIR might be a workaround rather than a fix. Consequently, trying to determine the underlying cause of an issue before removing the directory is a good idea.

- Understanding how a feature is used in practice within existing recipes can be very helpful. It is recommended that you configure some method that allows you to quickly search through files.

  Using GNU Grep, you can use the following shell function to recursively search through common recipe-related files, skipping binary files, `.git` directories, and the Build Directory (assuming its name starts with “build”):

  ```
  g() {
      grep -Ir \
           --exclude-dir=.git \
           --exclude-dir='build*' \
           --include='*.bb*' \
           --include='*.inc*' \
           --include='*.conf*' \
           --include='*.py*' \
           "$@"
  }
  ```

  Following are some usage examples:

  ```
  $ g FOO # Search recursively for "FOO"
  $ g -i foo # Search recursively for "foo", ignoring case
  $ g -w FOO # Search recursively for "FOO" as a word, ignoring e.g. "FOOBAR"
  ```

If figuring out how some feature works requires a lot of searching, it might indicate that the documentation should be extended or improved. In such cases, consider filing a documentation bug using the Yocto Project implementation of Bugzilla. For information on how to submit a bug against the Yocto Project, see the Yocto Project Bugzilla wiki page and the "Reporting a Defect Against the Yocto Project and OpenEmbedded" section.

Note

The manuals might not be the right place to document variables that are purely internal and have a limited scope (e.g. internal variables used to implement a single `.bbclass` file).