Alcatel-Lucent nmake Eclipse Plugin

September 2012

Table of Contents

List of Figures

1. Overview

1.1. Alcatel-Lucent nmake Eclipse CDT plugin

The Alcatel-Lucent nmake Eclipse CDT plugin supports use of Alcatel-Lucent nmake with Eclipse™ CDT. The plugin currently provides managed build definitions that configure CDT for nmake builds for Makefile-based projects. The definitions leverage nmake “common actions” to pre-populate builder options with targets useful across a broad range of nmake-based projects. Out of the box, the plugin provides an nmake specific CDT Project Type and CDT build configurations for Debug and Release builds on several platforms. In common cases, the build definitions substantially reduce the need to manually specify build options for nmake based builds.

Other features provided by the plugin include:

- A set of Eclipse Error Parsers which recognize nmake messages in the build output, classify their severity, and enable Eclipse to use them to for various purposes such as populating the problems view and creating problem markers.
- C and C++ sample projects packaged as Eclipse CDT project templates. For the sample projects provided, the need for manual option setup is eliminated.

The procedures described here were developed for Eclipse 4.2, CDT 8.1, and nmake 14 on Linux® and Oracle® Solaris and support the gcc C and C++ tool chains. See the nmake Eclipse plugin download page [http://www.bell-labs.com/project/nmake/download/eclipse/] for up-to-date version compatibility information.

1.2. About This Document

The following typographical styles are used in the document. These items are derived from the element classification scheme used in DocBook [http://docbook.org/]:

Style	**Used For**
command	The name of a command
`envvar`	An environment variable
error message	An error message
error type	The type of an error
`filename`	The name of a file or directory
GUI button	The text on a button in a GUI
GUI icon	Graphic and/or text appearing as an icon in a GUI
GUI label	The text of a label in a GUI
GUI menu	The name of a menu in a GUI
GUI menu item	The name of a menu item in a GUI
GUI sub menu	The name of a submenu in a GUI
`output`	Program output
`replaceable`	Content that may be replaced by the user
`user input`	User input

2. Getting Started

2.1. Installing Eclipse

You can download the latest Eclipse from the Eclipse download page [http://www.eclipse.org/downloads/]. The C/C++ development plugin (CDT) is required; you can start from the Eclipse Classic package and then install the CDT on top of it; or start with the Eclipse IDE for C/C++ Developers package which has CDT already included.

2.2. Installing the Plugin

You can install the nmake CDT plugin from the nmake Eclipse update site. See the nmake Eclipse support page [http://www.bell-labs.com/project/nmake/manual/eclipsesupport.html] for the latest information on installing the plugin and for the location of the update site.

1. Select **Help** > **Install New Software...**.

2. Click **Add...**.

3. In the **Add Repository** dialog, enter **`Alcatel-Lucent nmake Update Site`** in the **Name** field. Enter **`http://www.bell-labs.com/project/nmake/download/eclipse`** in the **Location** field. Click **OK**.

4. Back in the **Install** screen, select **Alcatel-Lucent nmake Update Site** from the drop down menu for the **Work with** field. An expandable list of available Alcatel-Lucent nmake plugins with version numbers should appear in the central display area. Expand the list and / or change display filter options at the bottom of the screen to display the Alcatel-Lucent nmake plugin entries.

5. Check the check box corresponding to the desired version of the Alcatel-Lucent nmake plugin.

6. Click **Next**. Click through the following screens to complete installation.

The nmake Alcatel-Lucent CDT support feature should now be installed.

3. Tasks

3.1. Project Build Setup

First verify that your project successfully builds standalone, outside of Eclipse. This will involve:

- **cd** to your project node root directory, say *`proot`*. Typically, *`proot`* will be the directory containing your `src`, `bin`, and `lib`. Since Eclipse defaults to building in the top level directory, it is easiest if *`proot`* is also the top level build directory, containing the top level `Makefile`. See Section 5, “Examples Guide” for a sample project organization.

- Set up your build environment: put **nmake** on your `PATH`. Make sure that the project node root is first on your `VPATH`. For a single node build, set `VPATH` to *`proot`*.

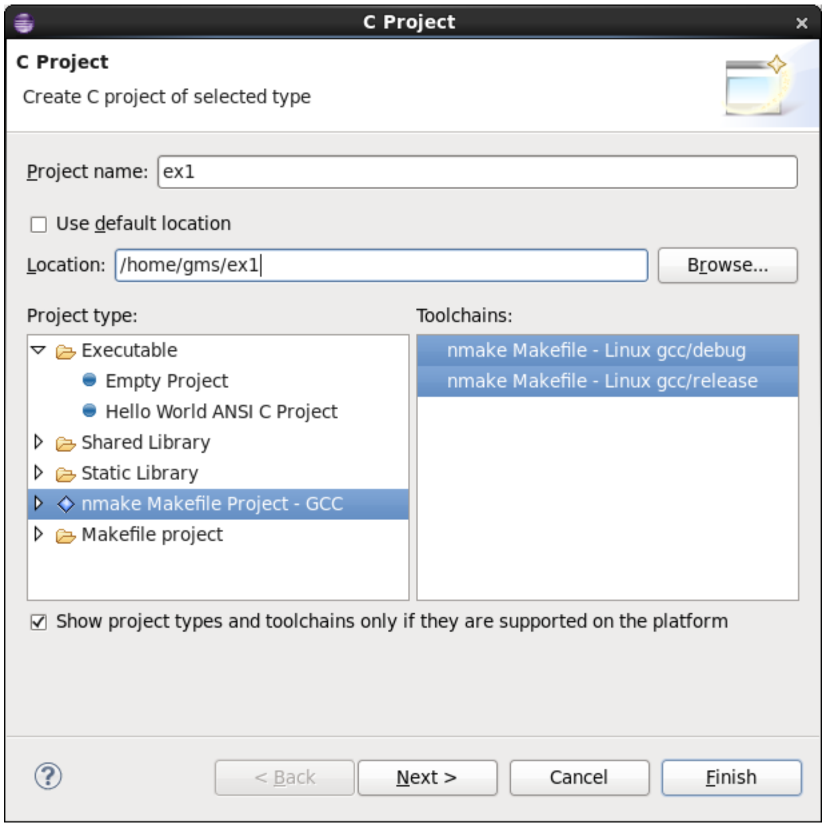
- Run a top level project build, typically via **nmake install**. Verify the build completes successfully. Clean up, typically via **nmake recurse clobber.install clobber**.

We recommend setting `recurse_begin_message = gnu` in the project global `Makefile` to configure nmake to output GNU make style recursion messages. This allows the Eclipse CDT error parser to track the current directory in recursive builds. See Section 5, “Examples Guide” for an example. By default, the build definitions provided in the nmake CDT plugin also set `recurse_begin_message=gnu` on the nmake command line. (The `recurse_begin_message = gnu` feature was introduced in nmake alu3.9.)

3.2. Eclipse Project Setup

Set up your environment for top level project builds as described above, then start Eclipse. If a workspace selection dialog pops up before the main Eclipse window opens, keeping the default workspace location is usually fine. (You might want to use a separate workspace location for each different version of Eclipse you want to be able to run.) Turn off automatic project builds by deselecting **Project** > **Build Automatically**. Create a new CDT project using **File** > **New** > **Project**, select **C Project** or **C++ Project** and click **Next**. Enter your **Project name**. Uncheck **Use default location** and set the **Location** field to the existing project root location *`proot`*. Select **nmake Makefile Project - GCC** under **Project type**. nmake toolchains with names similar to **nmake Makefile - Linux gcc/debug** and **nmake Makefile - Linux gcc/release** should then appear in the column labeled **Toolchain**. The toolchains are platform specific—by default, only toolchains supported on the current platform are displayed. These project types and toolchains are specific to nmake and were contributed by the nmake CDT plugin. As shown in the following screenshot, select both toolchains (select one and then include the other using shift-click). Click **Finish**. Answer **Yes** if the project setup wizard offers to open the C/C++ perspective.

Figure 1. CDT project creation wizard—project type and toolchain selection.



Project name: ex1

☐ Use default location

Location: /home/gms/ex1

Project type:
- Executable
  - Empty Project
  - Hello World ANSI C Project
- Shared Library
- Static Library
- nmake Makefile Project - GCC
- Makefile project

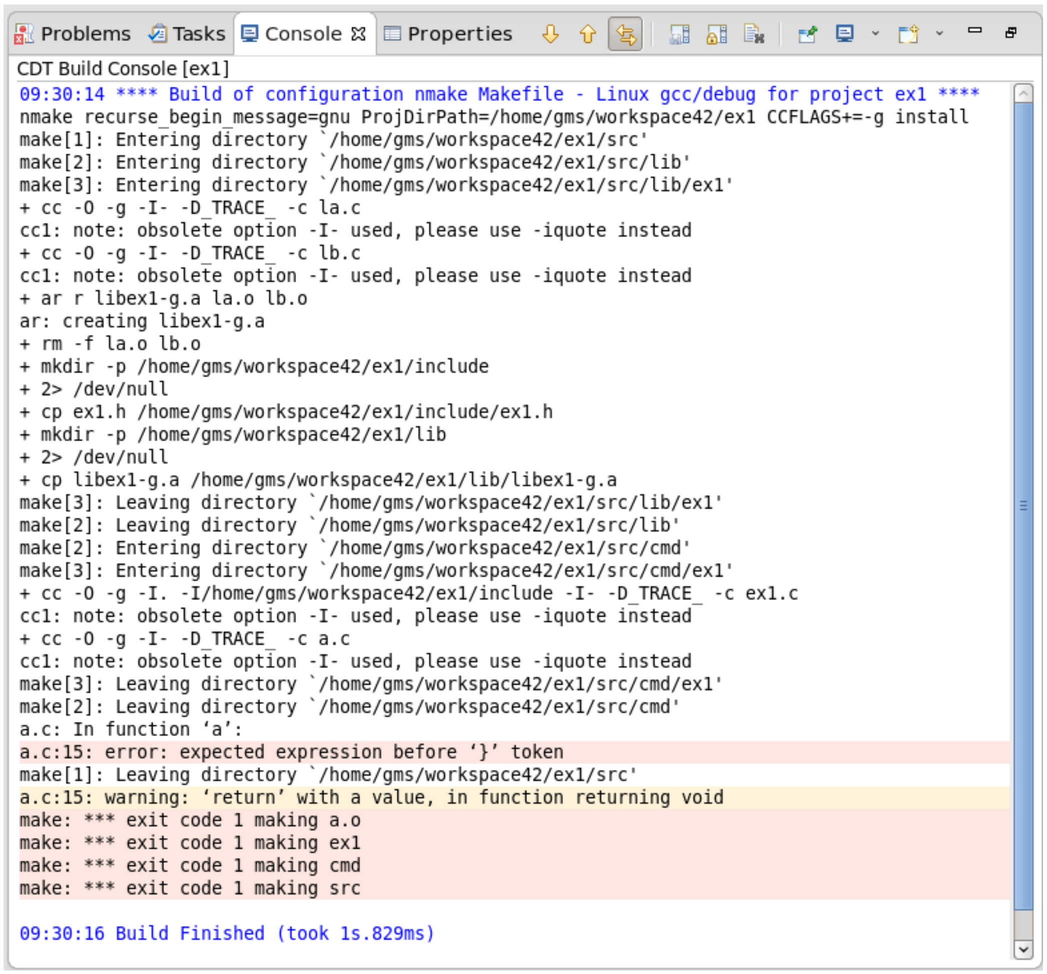
Toolchains:
- nmake Makefile - Linux gcc/debug
- nmake Makefile - Linux gcc/release

☑ Show project types and toolchains only if they are supported on the platform

3.3. Eclipse Project Builds

Select the project in the **Project Explorer** view, then build using **Project** > **Build Project**. This launches a top-level project build for the active configuration using **nmake install**. Build log output displays in the console view as seen below, in this case showing a debug build. Any errors and warnings from the C compiler and nmake are highlighted in the console build log. Problem markers are also created (viewable in the **Problems** view) that, if possible, map back back to the source file location where the error occurred.

Figure 2. Console build log for a project build.



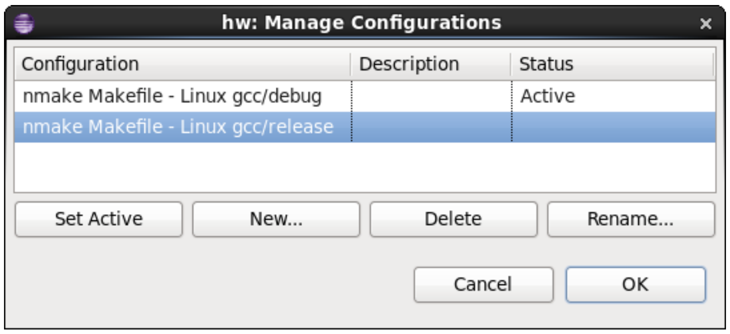
```
09:30:14 **** Build of configuration nmake Makefile - Linux gcc/debug for project ex1 ****
nmake recurse_begin_message=gnu ProjDirPath=/home/gms/workspace42/ex1 CCFLAGS+=-g install
make[1]: Entering directory `/home/gms/workspace42/ex1/src'
make[2]: Entering directory `/home/gms/workspace42/ex1/src/lib'
make[3]: Entering directory `/home/gms/workspace42/ex1/src/lib/ex1'
+ cc -O -g -I- -D_TRACE_ -c la.c
cc1: note: obsolete option -I- used, please use -iquote instead
+ cc -O -g -I- -D_TRACE_ -c lb.c
cc1: note: obsolete option -I- used, please use -iquote instead
+ ar r libex1-g.a la.o lb.o
ar: creating libex1-g.a
+ rm -f la.o lb.o
+ mkdir -p /home/gms/workspace42/ex1/include
+ 2> /dev/null
+ cp ex1.h /home/gms/workspace42/ex1/include/ex1.h
+ mkdir -p /home/gms/workspace42/ex1/lib
+ 2> /dev/null
+ cp libex1-g.a /home/gms/workspace42/ex1/lib/libex1-g.a
make[3]: Leaving directory `/home/gms/workspace42/ex1/src/lib/ex1'
make[2]: Leaving directory `/home/gms/workspace42/ex1/src/lib'
make[2]: Entering directory `/home/gms/workspace42/ex1/src/cmd'
make[3]: Entering directory `/home/gms/workspace42/ex1/src/cmd/ex1'
+ cc -O -g -I. -I/home/gms/workspace42/ex1/include -I- -D_TRACE_ -c ex1.c
cc1: note: obsolete option -I- used, please use -iquote instead
+ cc -O -g -I- -D_TRACE_ -c a.c
cc1: note: obsolete option -I- used, please use -iquote instead
make[3]: Leaving directory `/home/gms/workspace42/ex1/src/cmd/ex1'
make[2]: Leaving directory `/home/gms/workspace42/ex1/src/cmd'
a.c: In function ‘a’:
a.c:15: error: expected expression before ‘}’ token
make[1]: Leaving directory `/home/gms/workspace42/ex1/src'
a.c:15: warning: ‘return’ with a value, in function returning void
make: *** exit code 1 making a.o
make: *** exit code 1 making ex1
make: *** exit code 1 making cmd
make: *** exit code 1 making src

09:30:16 Build Finished (took 1s.829ms)
```

You can clean up by doing **Project** > **Clean...**, which runs the command **nmake recurse clobber clobber.install**. Uncheck **Start a build immediately** unless you want to rebuild immediately after clobbering.

The above procedure generates objects compiled for debugging. To generate objects compiled for release, change the active configuration by selecting the project in the **Project Explorer**, then select **Project** > **Build Configurations** > **Set Active** > **nmake Makefile - Linux gcc/release** (or the equivalent for your platform), and rebuild the project. Alternatvely, select **Manage...** to bring up the **Manage Configurations** dialog and select the desired build configuration as shown in the following screenshot:

Figure 3. Manage build configurations dialog.


hw: Manage Configurations

Configuration	Description	Status
nmake Makefile - Linux gcc/debug		Active
nmake Makefile - Linux gcc/release		

3.4. Custom Make targets

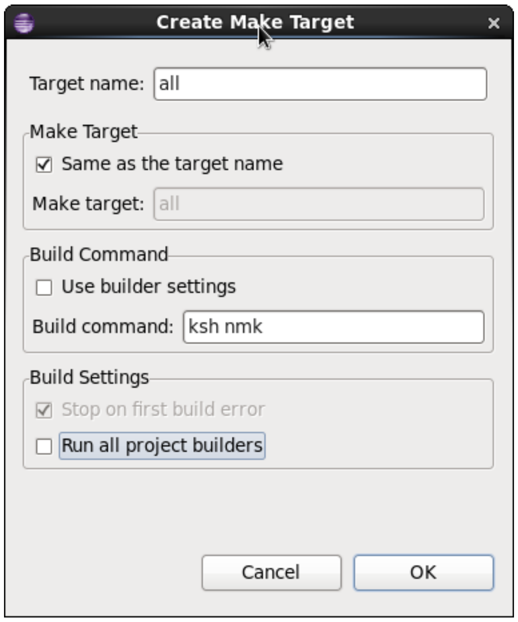
In addition to the project level build definitions, CDT allows you to define custom make targets that may appear at any level in the build hierarchy. Custom make targets may be defined by right clicking on any Makefile in your build hierarchy as displayed in the **Project Explorer** view to bring up the context menu. Select **Make Targets**, then **Create...** to bring up the **Create Make Target** dialog. With one exception, standard CDT procedures for defining custom make targets work for nmake. The exception is the setting for the **Build Command** field of the **Create Make target** dialog. When some versions of CDT run the specified build command, they do not set environment variable `PWD` appropriately for targets not at project root. This leads to incorrect values for nmake variables such as `VROOT`. A workaround is to execute nmake through a simple shell wrapper script. A sample script is shown in Example 1, “Sample nmake shell wrapper script”. In addition to wrapping nmake, the displayed script filters out unwanted compiler warnings.

Example 1. Sample nmake shell wrapper script

```
(
  set -x
  nmake $*
) 2>&1 | grep -v 'please use -iquote instead'
echo end $0 $*
```

To use the script, place the it in a file called `nmk` in a directory on your `PATH` and use **ksh nmk** instead of **nmake** in the **Build Command** field when creating a custom make target:

Figure 4. Custom make target creation dialog.



Target name: all

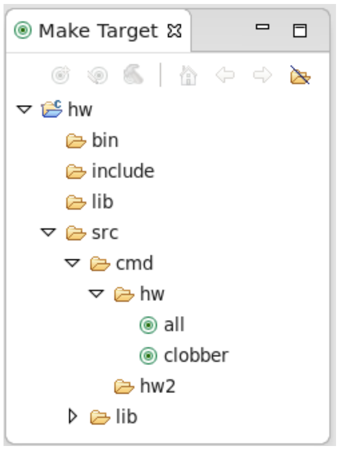
Make Target
☑ Same as the target name
Make target: all

Build Command
☐ Use builder settings
Build command: ksh nmk

Build Settings
☑ Stop on first build error
☐ Run all project builders

Custom make targets display in the **Make Targets** view:

Figure 5. Custom make targets view.



Double click on the target icon to build the target, or select the target and click the target toolbar icon.

4. Reference

4.1. nmake Build Definitions

The build definitions component of the nmake CDT plugin provides CDT build model element definitions that configure Eclipse CDT for nmake Makefile based builds. The definitions provide an nmake-specific project type called `nmake Makefile Project - gcc` containing 2 nmake-specific build configurations, called `nmake Makefile - Linux gcc/debug` and `nmake Makefile - Linux gcc/release` (or the equivalent for your platform). Each configuration provides a toolchain definition specialized for its purpose—debug or release build. Each toolchain provides a `Builder` element as well as individual tools including a preprocessor, compiler, assembler, and linker. The `Builder` definitions specify nmake arguments appropriate for debug and release builds, respectively. The tool definitions essentially serve as containers for information needed to adjust error parser, binary parser, and scanner discovery profile settings needed to correctly set up error parsers, paths and symbols. These settings allow the CDT indexer to properly parse input to set up the internal model of the code under development. The tool definitions all inherit from platform base tool definitions. This approach maximizes reuse of platform definitions and makes the system more robust to toolchain definition changes in new releases of CDT.

CDT does not support option enablement for builders. This forces definition of separate toolchains for each configuration. However, the toolchain definitions were able to factor out common nmake `Builder` definitions using inheritance and derive debug/release versions from those. This allows common nmake `Builder` setting definitions to be factored out in one place, and allows the definitions to automatically acquire updated definitions in new platform releases.

When the plugin is installed, the nmake build definitions extend the CDT build model and appear in the CDT GUI as options in the project setup wizard and in the CDT properties view. Selection of these elements configures the CDT project with build model elements specialized for nmake.

The provided build elements and definitions leverage nmake baserules conventions including standardized variables and common actions. For examples, project level builds are performed using `install` and cleans are performed using `recurse clobber.install clobber`. Since nmake based projects inherit these rules “for free”, these canned definitions should work for the majority of nmake based projects. However, the build definitions just provide initial default settings, project-specific customization is easily performed using the CDT GUI.

The definitions add `recurse_begin_message=gnu` to the nmake command line. This configures nmake to output recursion messages compatible with existing CDT make output parsers. We recommend setting `recurse_begin_message=gnu` in the project global `Makefile`. Setting it on the command line provides additional assurance that this parameter gets appropriately set.

Note that it was not possible to set the `Build Location` parameter in the build definitions. Since the build location defaults to the top level project directory, it is easiest to run project builds from that location. Section 5, “Examples Guide” includes a top level `Makefile` that simply recurses to the `src` directory for the actual build. Alternatively, the build location is configurable through the CDT GUI.

4.2. Error Parsers

This feature implements nmake specific Eclipse error parsers for Eclipse CDT. These recognize diagnostic messages generated by nmake in the build output and create an Eclipse “problem marker” for each message found. Each problem marker has a description, creation time, severity, file name, file path, and line number. Severity can be one of `Error`, `Warning`, `Info`, or `Ignore`. File name, path, line number, and description can be derived from the message text.

Problem markers are displayed in the Eclipse “Problems View”, where problem severity is indicated by an icon, and the view can be sorted and filtered by various criteria. Problem markers associated with a file / line number are displayed in the left margin of editors with the right margin displaying an overview of all problems in the file. Matching lines in the console build log are highlighted as indicating a problem. Also, files in the Project Explorer are marked with an error icon if they contain 1 or more problems, and folders containing files with problems are also marked, all the way up to the project root. Clicking on a problem in the problem view or a highlighted line in the console build log brings up an editor positioned at the problem location.

The provided error parsers classify nmake error dianostic messages into 5 types and provide a recognizer for each type. A typical diagnostic message looks like

```
make: "Makefile", line 17: warning: file does not end with newline
```

This message would be recognized by the nmake error parsers as a type 1 diagnostic, with file set to `Makefile`, line set to `17`, and severity set to `Warning`. Type 2 diagnostics are similar to type 1 but indicate an error, types 3 and 4 indicate warnings and errors, respectively, with file and line unspecified. Type 5 diagnostics indicate are generated by the nmake license manager. Diagnostics without a specified file / line number are associated with the project as a whole rather than on a specific resource within the project.

In certain cases, error and warning diagnostics do not indicate the correct file and line number. Specifically, run time nmake messages orginating when inside a `.MAKE` block may provide the target name in the file position, and the line number of the block disregarding commands and white space. These cases are currently treated like a message without a file name / line number (the CDT error parser framework treats the errors as such when the provided file name does not correspond to an existing file).

In addition to defining nmake specific error parsers, the nmake specific build definitions (see Section 4.1, “nmake Build Definitions”) have been modified so that the nmake error parser is automatically enabled when the nmake build definitions are selected. The CDT pushd/popd current directory detector is also prepended to the list of error parsers in the build definition. This enables interpretation of GNU style recurse messages, allowing error markers created from problems in recursive builds to be associated with files in the correct project subdirectory (nmake can be configured to output GNU style recurse messages by setting `recurse_begin_message=gnu`). The gmake error parser is also added in case somewhere along the line a gmake or make style error message needs to be interpreted in an nmake based build.

5. Examples Guide

5.1. Overview

This section describes an example nmake C Makefile project delivered with the nmake CDT plugin. The example is packaged as a CDT project template, simplifying example installation and setup. The example project has several purposes:

- It provides a working example showing interoperability of Eclipse with nmake.
- Since it is packaged as a CDT template, it provides a simplified procedure for creating a new nmake-based Eclipse CDT project. You can customize this new project for your own needs.
- It demonstrates how to go about building a typical existing multi-level nmake Makefile-based project in Eclipse CDT.
- It also provides an example of a recommended approach for organizing a multiple directory nmake project that builds several commands and libraries.

You may assign a name of your choice to the example project during project creation. The following descriptions assume that the project name is `hw` (for “Hello World”).

An nmake C++ Makefile project template is also provided in the nmake CDT plugin. Use of the C++ project template is similar to that of the C project template.

5.2. Description of Example

The nmake C project template provides a simple but realistic project example. The project builds 2 commands and 1 library, with the source for each located within its own sub-directory:

```
hw/
    Makefile
    src/
        Makefile
        global.mk
        cmd/
            Makefile
            hw/
                Makefile
                hw.c
                a.c
                b.c
                b.h
            hw2/
                Makefile
                hw2.c
                a.c
        lib/
            Makefile
            hw/
                Makefile
                lb.c
                la.c
                hw.h
    bin/
    include/
    lib/
```

This project is representative in that it demonstrates command and library builds, recursive makes, dependencies on installed headers/libraries, use of project level global Makefiles, installs, and clobbers/cleans. The example has characteristics of larger projects and the structure is scalable. It can support complex requirements such as separation of architecture-dependent generated targets (for example, it could easily generate Solaris and Linux objects from the same source and cleanly separate the generated objects through viewpathing.) As shown below, this project structure leads to simple declarative Makefiles that provide full featured build functionality.

`hw` is the project root (in nmake terminology, the viewpath node root) containing the `src`, `bin`, `include`, and `lib` directories. `src` is the root of the source tree; generated executables and libraries are built locally and installed under `bin`, `include`, and `lib`. `global.mk` is the project global Makefile and is included in all project Makefiles. Following are contents of selected Makefiles.

`hw/src/global.mk`:

```
/* generate GNU style make recurse messages */
recurse_begin_message = gnu

/* install build products in build node root directory */
INSTALLROOT = $(ProjDirPath)

/* example project-wide header directory */
.SOURCE.h : $(INCLUDEDIR)

/* example project-wide library directory */
.SOURCE.a : $(LIBDIR)
```

`hw/src/Makefile`:

```
include ../src/global.mk

:MAKE: lib - cmd
```

`hw/src/cmd/Makefile`:

```
include ../../src/global.mk

:MAKE: *
```

`hw/src/cmd/hw/Makefile`:

```
include ../../../src/global.mk

hw :: hw.c a.c b.c -lhw
```

Complete source for this project is available in the provided template project.

`hw/src/global.mk` is a project level global Makefile that is included by all other Makefiles. It contains project settings visible to all project Makefiles. Setting `recurse_begin_message = gnu` configures nmake to output GNU make style recursion messages. This allows a CDT supplied error parser to track the current directory in recursive builds. The default nmake supplied build definitions also set `recurse_begin_message = gnu` on the default nmake command line as a convenience for existing projects that might not have `recurse_begin_message` set appropriately in their own Makefiles. (The `recurse_begin_message = gnu` feature was introduced in nmake alu3.9.) See Section 5.5, “Notes on the Example Project” for an explanation of `ProjDirPath`. The `:MAKE:` assertion tells nmake to recurse to the directories specified on the right hand side of the assertion. `hw/src/cmd/hw/Makefile` is a leaf Makefile and specifies how to build command **hw**.

5.3. Creating the Example Project

nmake should be on your `PATH` before starting Eclipse. In addition, a C compiler called **cc** should be on your `PATH`, alternatively point environment variable `CC` to your C compiler (for example, **export CC=/usr/bin/gcc**). Once in Eclipse, turn off automatic project builds by selecting **Project** > **Build Automatically**.

1. Select **File** > **New** > **Project...** > **C/C++** > **C Project**.

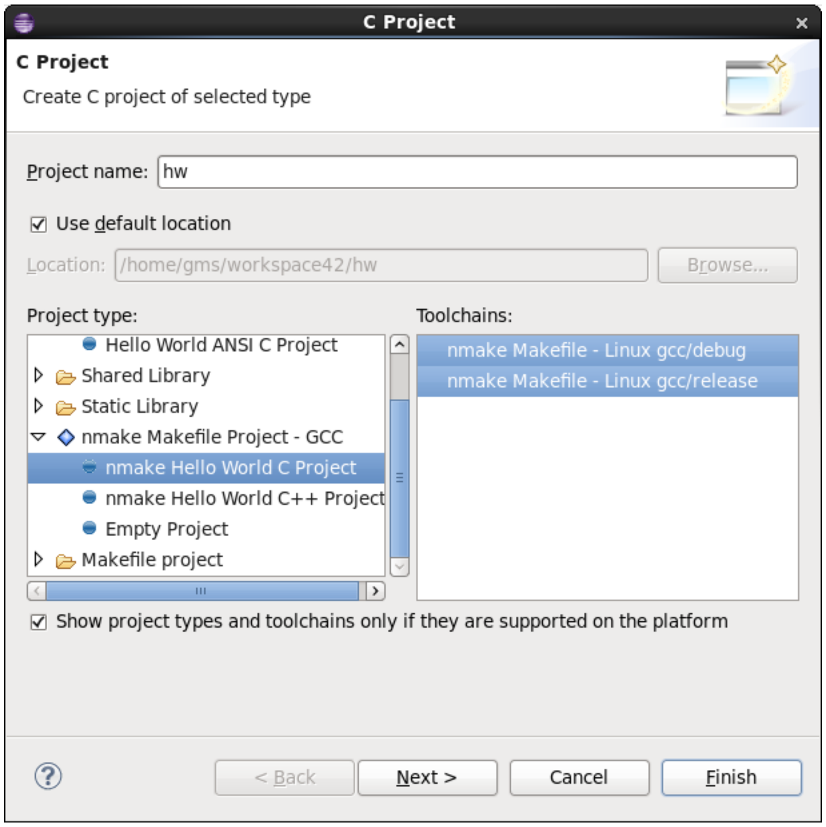
2. Click **Next**.

3. Enter a project name, for example, **`hw`**. This will become the basename of various elements of the generated example project, such as commands and libraries. Consequently, you probably want to avoid spaces in the project name.

4. Select **nmake Makefile Project - GCC** > **nmake Hello World C Project**.

5. Select both toolchains **nmake Makefile - Linux gcc/debug** and **nmake Makefile - Linux gcc/release**, or the equivalent for your platform (use shift-click to add a toolchain to existing selection.)

Figure 6. CDT project creation wizard—template project type and toolchain selection.


C Project

C Project

Create C project of selected type

Project name: hw

☑ Use default location

Location: /home/gms/workspace42/hw

Project type:
- Hello World ANSI C Project
- Shared Library
- Static Library
- nmake Makefile Project - GCC
  - nmake Hello World C Project
  - nmake Hello World C++ Project
  - Empty Project
- Makefile project

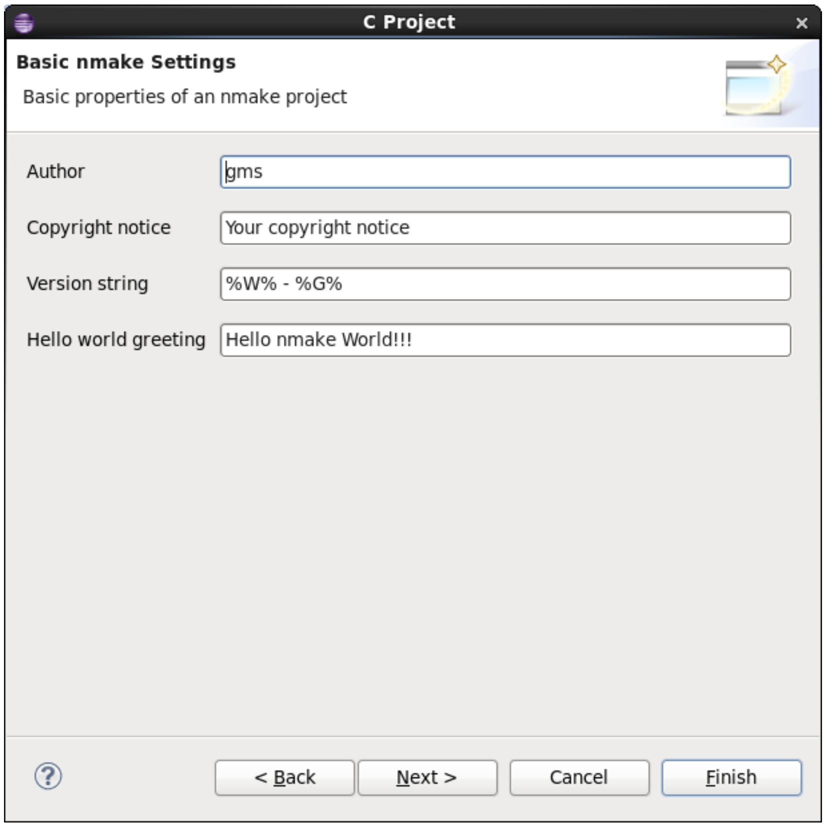
Toolchains:
- nmake Makefile - Linux gcc/debug
- nmake Makefile - Linux gcc/release

☑ Show project types and toolchains only if they are supported on the platform

6. Click **Next**.

7. Enter properties for the example project. These will appear in the generated project as appropriate, such as in generated source file prologue comments.

Figure 7. CDT project creation wizard—template properties dialog.



Author: gms
Copyright notice: Your copyright notice
Version string: %W% - %G%
Hello world greeting: Hello nmake World!!!

8. Click **Next**, then **Finish**.

9. Click **Yes** if Eclipse offers to open the C/C++ perspective.

10.A new project using the supplied project name should appear in the **Project Explorer**.

5.4. Building the Example Project

5.4.1. Building within Eclipse

All of the following build procedures require the example project to be selected before initiating a build operation. Select the project before each procedure by selecting its name in the **Project Explorer**.

- To build the project using the default debug configuration, select **Project** > **Build Project**. The entire project should build with the build log appearing in the **Console** view.
- To clean the project, select **Project** > **Clean...** > **Clean projects selected below**. Select the example project. Deselect **Start a build immediately**. Click **OK**.
- To build the project using the release build configuration, select **Project** > **Build Configurations** > **Set Active** > **nmake Makefile - Linux gcc/release** (or the equivalent for your platform). Build the project using **Project** > **Build Project**. The project should build using release compilation options.

5.4.2. Standalone Builds (Outside of Eclipse)

1. Start a console terminal and **cd** to the top level project directory `hw`.

2. Set the `ProjDirPath` environment variable to the current directory. For example, if using **ksh** type **export ProjDirPath=$PWD**.
3. Type **nmake install** to build and install the entire project. Each library and executable is built locally and then installed in the project install directories `lib` and `bin`, with required include files going to `include`.
4. Type **nmake recurse clobber.install clobber** to remove generated targets in each local development directory and all installed targets.

5.5. Notes on the Example Project

- The template project is generic but can be customized for needs of specific development projects. For example, the source header format is easily modified and the set of substitutable fields can be easily changed.
- The current template avoids the use of `VPATH` (and therefore `VROOT`) in order to simplify initial project configuration. For now, the value of the CDT variable `ProjDirPath` is passed down to nmake as a way to find the project root. This may change in later releases.

6. Notes

- Eclipse CDT is not yet viewpath aware, so files down the viewpath will not show up in the project view. For now, files down the viewpath may be brought into the top viewpath node for editing.
- The plugin does not currently provide an nmake Makefile editor, so certain constructs unique to nmake Makefiles get error markers when editing using the existing CDT Makefile editor. These markers are harmless.

7. Tips and Tricks

7.1. General Tips

7.1.1. Running Eclipse CDT

Eclipse CDT is designed to work with local C/C++ projects, where Eclipse is running on the machine where the project is located. Furthermore, Eclipse is most responsive when run on the desktop. Since the nmake CDT plugin is designed to support Linux/Unix® based projects, this would imply running a native desktop Linux/Unix, or a virtualized Linux/Unix using a virtual machine monitor such as Virtual Box or VMware® workstation running under hosts such as Windows® systems. We have had excellent results with both of these approaches.

Another popular approach is to run Eclipse remotely over VNC.

Recently there have been a number of projects aimed at providing a remote development capability, allowing the IDE to run locally with projects located remotely. There are several approaches, each with its own set of tradeoffs. Use of nmake in remote development scenarios is currently under evaluation.

7.1.2. Project Location

For projects with an existing build node that already builds outside of Eclipse, we recommend using the build node as project location when creating a CDT project. This will typically be a location in the file system outside of the Eclipse workspace. The workspace then in effect becomes just a container for project meta-information.

7.1.3. Subdirectory Makes

An alternative to use of CDT make targets for subdirectory makes is to **cd** to the target directory and run nmake directly from an nmake wrapper script run using the top level project build command.

7.2. Troubleshooting

7.2.1. Wrong Error Locations

Symptom: Build errors in `stderr` are correctly recognized (and colorized, etc.), but not always associated to the correct file in the correct project subdirectory.

Tip: This occurs when messages in `stderr` are not bracketed within the correct recurse push/pop messages in the CDT build console log. This is apparently due to stream buffering issues in the CDT build console since a workaround is to change the build command from **nmake ...** to **ksh -c 'nmake ... 2>&1'**. This forces `stderr` into the same output stream as `stdout` before it gets to CDT.

7.2.2. Unexpected Builds

Symptom: Builds are launched unexpectedly during project creation.

Tip: Turn off **Project** > **Build Automatically** before creating a new CDT C/C++ project. Otherwise, Eclipse will kick off a build immediately upon project creation and after resource saves, probably not what you want.

7.2.3. Install step fails in sample project

Symptom: In the supplied sample project, nmake is unable to locate or create the `include` install directory during the install step.

Tip: Check that the `ProjDirPath` variable is set to the project root directory on the generated nmake line, which should be visible in the **Console** view build log after a build. If `ProjDirPath` is not set, it may mean that a resource in the sample project had not been selected prior to launching the build.

7.2.4. Unable to suppress build after a clean

Symptom: The **Start a build immediately** checkbox does not appear in the **Project** > **Clean...** dialog.

Tip: Verify that **Project** > **Build Automatically** has been turned off.

8. What's New

Version 1.0.0 released September, 2012. New features include an nmake error parser, auto-enablement of the nmake error parser in the nmake builder definitions, and revamped and updated plugin documentation available in multiple formats. Productization improvements include moving to the Eclipse standard major.minor.service.qualifier plugin version numbering scheme for improved interoperability with the Eclipse p2 provisioning system, plugin signing for improved verification during plugin installation, and updated branding elements. Under the hood, the plugin release builds move to a fully automated, “headless”, build process against a controlled target platform, leading to higher quality and repeatable product. The plugin supports the Eclipse 4.2 (Juno)/CDT 8.1 release and has been tested on Linux and Solaris.

Version 0.5.0 released July, 2009. Update dependencies allowing install with Eclipse 3.4+. Workaround CDT 6.0 bug which caused nmake project templates to not display. Eclipse variable `${project_loc}` was not getting set leading to failure to build nmake template projects. Changed to use CDT variable `${ProjDirPath}` instead. Update default version strings in project template definition files. Verify operation with Eclipse 3.5/CDT 6.0.

Version 0.4.0 released March, 2008. Introduce build definition toolchain support for Solaris. Introduce support for C++, including a new C++ project template. Globally change `alu` to `alcatel_lucent` in all identifiers and names. Pull out key build commands into property file for easier maintenance/ modification. Add an empty project template for the nmake project type. Update help book documentation. Miscellaneous cleanups.

Version 0.3.2 released December, 2007. Includes build definitions for gcc/Linux, help documentation, and example project packaged as a CDT template.

9. Legal

For more information regarding Alcatel-Lucent nmake Product Builder, including current release information, downloads and technical support, visit our web site at http://www.bell-labs.com/project/ nmake/.