Embedix SDK 2.4 Tools

Disclaimer, Trademarks, and Copyright Information

Lineo, Inc. makes no representations or warranties with respect to the contents or use of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Lineo, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Lineo, Inc. makes no representations or warranties with respect to any Lineo software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Lineo, Inc. reserves the right to make changes to any and all parts of Lineo software, at any time, without any obligation to notify any person or entity of such changes.

Lineo and Embedix are registered trademarks of Lineo, Inc. The stylized Lineo logo is a trademark of Lineo, Inc. The Metrowerks name and logo as well as the CodeWarrior name and logo are copyright Metrowerks, a Motorola, Inc. company.

Other product and company names mentioned in this document may be the trademarks or registered trademarks of their respective owners.

Copyright © 2002 Lineo, Inc. All rights reserved. No part of this publication may be reproduced, photocopied, stored on a retrieval system, or transmitted without the express written consent of the publisher.

Lineo, Inc.
588 West 400 South
Suite 150
Lindon, UT 84042 USA
http://www.lineo.com

Embedix SDK 2.4 Tools
Part Number: EMBD-SDK-TOOLS-0502
May 2002

Contents

PREFACE **About This Guide** ... v

Conventions Used in This Document ... v
Admonitions ... v
Key Combinations ... vi
Special Fonts and Capitalization ... vi
Additional Resources ... vii

CHAPTER 1 **Introduction to SDK Tools** ... 1

Embedix Target Wizard ... 1
Package Editor ... 2
Metrowerks CodeWarrior for Lineo Embedix SDK ... 2
Data Display Debugger (DDD) ... 2
Embedix GPL Compliance Toolset ... 3
Graphical Remote Process Analyzer ... 3
Helper Utilities ... 3

CHAPTER 2 **Packaging with Package Editor** ... 5

What is an Embedix Package? ... 6
Starting Package Editor ... 7
From the Menu Bar ... 7
From the Command Line ... 7
Exploring the Interface ... 9
Menus ... 11
Shortcut Icons ... 11
Tabs ... 12
Log Window ... 12
Overview to Creating a Package ... 12
General Packaging Steps ... 13
Tutorial: Creating a New Package ... 13

Tutorial: Creating a Package for an Application from an External Source (like Open Source Applications) 19
Tutorial: Creating Binary Packages to Use in Target Wizard 21
Downloading the Source 25
Unpacking the Source 25
Modifying LBC Sections 26
What is an LBC? 26
LBC File Tab 27
LBC File Sections 30
LBC File Inheritance 32
Build Variables in LBC Files 33
Modifying ECD Files 37
What is an ECD? 37
ECD File Tab 38
Why and How to Use ECD 44
Build Variables in ECD Files 48
Modifying Source Files 51
Source Tab 51
Using an External Text Editor 51
Other Options 52
Viewing "Diffs" and Making Patches 52
Installing Patches from External Sources 53
Building the Binary Image 54
Installing the Package 56
Distributing Packages as LPF Files 57
Using the Embedix Tool Chains 58

CHAPTER 3 **Configuring & Using Metrowerks CodeWarrior with Embedix SDK** 61
Configuring CodeWarrior for Lineo Embedix SDK 2.x 61
Embedix SDK Installation Options 61
Post-install SDK Configurations 63
CodeWarrior Installation and Initial Configuration 64
Common Global Configuration Options for CodeWarrior 66
Recording Embedix SDK Tools Setting 67
CodeWarrior Per Project Configuration Options 71
Library and Include Files 72
Output Directory 73
Useful Output Directory Locations: 74

Debugger 74
Compiler Command Line Arguments 76
Linker 79
Configure CodeWarrior to Use the Right GNU Tools 80
Configure CodeWarrior to Recognize *.ecd and *.lbc Files .. 81
Creating Project “Stationery” Files 83
Adding New Tools to the CodeWarrior Menus 84
Using CodeWarrior with Embedix SDK 85

CHAPTER 4 **Debugging Using GDB and DDD** 95
Understanding Remote Debugging 95
Using DDD and GDB from Target Wizard's Tools Menu 97
Launching DDD 97
GDBs Used by the SDK 97
Manual Execution of DDD 98
Using DDD 98
Editor Configuration 100
Web Browser Configuration 100
Tip of the Day 101
Warning 101
Overview of Debugging an Embedix Target Image 101
Kernel Debugging at Boot Time 102
Kernel Debugging—Started from the Shell 105
Kernel Module Debugging 106
Application Debugging over IP 109
Application Debugging over Serial Port 111
Additional Resources 113

CHAPTER 5 **Helper Utilities** 115
Building 115
Privileges 116
Deployment 116
emb_build 117
emb_mkproj 118
suwrapper 119
suwrapper.conf 120
tcconfig 122

APPENDIX **Manual Method of Packaging** 125

What is an Embedix Package? 125
Creating an LBC 126
Buildcontrol File Features 126
Buildcontrol File Sections 126
Buildcontrol Inheritance 128
Build Phases 130
%pkg_file 130
%patches 130
%bld_dir_name 131
%cflags 131
%cfgopts 131
%spec 131
%bin 131
%bld_targ 131
%makep 132
%makec 133
%makerc 134
%makeb 134
%makei 134
%makest 135
%makedc 135
Creating an ECD 136
A Typical ECD File 136
Build Options in an ECD 143
Creating a Specpatch File 146
Summary 151
Using a Tarfile, SRPM, or CVS Directory For Source 151
Tarfile 152
SRPM 152
CVS Repository 152
Creating an SRPM 153

INDEX 155

PREFACE

About This Guide

This preface includes information on formatting practices used throughout this Lineo® Embedix® document and a list of additional resources.

Conventions Used in This Document

The style conventions used in the printed and PDF format of this document do not necessarily apply to other formats. During conversion to HTML, some of these conventions may be lost.

This document uses the following graphical and typographical conventions:

- Admonitions
- Key combinations
- Special fonts and capitalization

Admonitions

Note, Tip, and Warning paragraphs draw your attention to additional information which may help you avoid losing data or time.

Note: Notes contain additional information about the current topic.

Tip: Tips contain suggestions that may save you time or effort.

Warning: Warnings contain critical information that you need to understand before proceeding. Ignoring information in a warning may cause loss of data or time.

Key Combinations

Key combinations (such as Ctrl+O) are presented throughout this document and should be used in the following way:

1. Press and hold the first key (such as Ctrl).
2. Press the second key (such as O).
3. Release both keys.

Special Fonts and Capitalization

In the printed or PDF version of this document, the following special fonts and capitalization rules apply:

- **Commands or user input:** All commands or data to be entered on an on-screen data entry line appear in **`bolded Courier`** font. This may include commands used with options, paths to directories or files, or other simple input, such as filenames.
- **Code or computer output:** CodeAny code sample, including command output, is shown in `Courier` font.
- **Capitalization:** Linux filenames and commands are case-sensitive. In most instances, they are lowercase. When you enter a filename or command, use the same case that appears in your instructions or examples.
- **On-screen buttons:** When procedures refer to a particular on-screen button, the name of the button appears in uppercase (such as “click SAVE”), regardless of how it appears on the screen.
- **Keyboard keys:** When procedures refer to a particular key on a keyboard, only the initial key is capitalized (such as “press Tab”), just as it appears on a U.S. standard keyboard. This also applies to key combinations.

Additional Resources

The following resources are available to provide you with additional support.

- *Embedix SDK Getting Started* (Linux hosted) or
  Embedix SDK for Windows Getting Started (Windows hosted)
- *Embedix SDK Target Wizard User Guide*
- *Embedix SDK Reference Manual*
- *Embedix RealTime Programming Guide*
- Lineo Support Web site:
  `http://www.lineo.com/support`

Note: Most printed manuals that ship with Lineo products are also available in PDF and HTML formats on the product CD-ROM.

CHAPTER 1 Introduction to SDK Tools

This chapter provides an introduction to the tools that are either included in this SDK product or available for purchase as an **Embedix SDK** or **Embedix SDK for Windows** add-on. The tools introduced in this chapter are:

Embedix Target Wizard
Package Editor
Metrowerks CodeWarrior for Lineo Embedix SDK
Data Display Debugger (DDD)
Embedix GPL Compliance Toolset
Graphical Remote Process Analyzer
Helper Utilities

Embedix Target Wizard

Embedix Target Wizard is the premiere tool included in the Embedix SDK and the Embedix SDK for Windows products. Target Wizard provides a graphical user interface that assists you in developing and deploying customized embedded Linux systems. It also helps you:

- Manage package dependencies
- Remove conflicts between components
- Integrate applications with the operating system
- Reduce the size of your embedded target image
- Cross compile for target
- Deploy to target

For a detailed introduction to Target Wizard and instructions on use, refer to the *Embedix SDK Target Wizard User's Guide*.

Package Editor

There are a number of methods you can use to add your custom applications to an Embedix target image. There are two methods we recommend: Merge your files into a target image or create custom Embedix packages.

For instructions on merging files into a Target Image, refer to the *Embedix SDK Target Wizard User's Guide*.

For instructions on creating custom Embedix packages, see "Packaging with Package Editor" on page 5.

Metrowerks CodeWarrior for Lineo Embedix SDK

Instructions for configuring this tool can be found in "Configuring & Using Metrowerks CodeWarrior with Embedix SDK" on page 61.

Data Display Debugger (DDD)

To assist you in debugging efforts, Embedix SDK includes the gdb and ddd debugging tools. You can use these to debug your target image after the target image has been deployed to the target.

The GNU Debugger, gdb, is a debugging tool that provides source-level run-time debugging. It is used during development to aid in finding and fixing problems.

The Data Display Debugger, ddd, is a graphical user interface for gdb, which most users enjoy using rather than using the command-line driven gdb by itself. The ddd interface provides menus for gdb functions and windows for gdb commands, code listing, and variables. It also automatically runs gdb commands in the background to provide useful information.

For instruction on using the Data Display Debugger, see "Configuring & Using Metrowerks CodeWarrior with Embedix SDK" on page 61.

Embedix GPL Compliance Toolset

The Embedix GPL Compliance Toolset is an Embedix SDK or Embedix SDK for Windows add-on product available to all SDK users for an additional fee. It is comprised of a collection of tools that, once installed, help you analyze your project packages for license compliance.

For information on purchasing this product, refer to the Lineo Web site http://www.lineo.com.

Graphical Remote Process Analyzer

Graphical Remote Process Analyzer in an Embedix SDK or Embedix SDK for Windows add-on product. It provides a tool named LTOP, which is a monitor that displays the status of the processes currently in existence. It is much like the Linux utilities top or ps, but with a graphical user interface.

LTOP can be used to monitor either the host system or a target system running the `ltop_target` client software. The graphical user interface uses the Qt toolkit.

For information on purchasing this product, refer to the Lineo Web site http://www.lineo.com.

Helper Utilities

The helper utilities are utilities that are available to Embedix SDK or Embedix SDK for Windows users that are external to Embedix Target Wizard. The correlating chapter in this book contains information on Building, Privileges, and Deployment, and includes a copy of a collection of related man pages.

For more information, see “Debugging Using GDB and DDD” on page 95.

CHAPTER 2 Packaging with Package Editor

Embedix® Target Wizard allows you to place custom applications and configuration files on your target. One method of doing this is to "package" the application or files and then add the package to your Target Wizard project. This allows you to configure and build your package along with the Embedix SDK packages.

In Embedix SDK 2.4 (or higher) or Embedix SDK for Windows 2.4 (or higher), Package Editor, a tool with a graphical user interface, makes packaging simple. It fulfills two purposes:

- **It makes modifying pre-existing packages easy.**
  To easily tweak existing packages until they work for a new architecture, a newer version of a package's source code, etc.
- **It facilitates the creation of new packages.**
  To reduce or eliminate the need to understand LBC and ECD files and to know the directory structure of projects, as is required when creating a package by hand.

Although creating a new package is briefly explained in "Overview to Creating a Package" on page 12, the primary focus of this document is to demonstrate how to create an Embedix SDK package from pre-existing source using the Package Editor.

This chapter provides information and instruction on:

"What is an Embedix Package?" on page 6
"Starting Package Editor" on page 7
"Exploring the Interface" on page 9
"Overview to Creating a Package" on page 12
"Downloading the Source" on page 25
"Unpacking the Source" on page 25
"Modifying LBC Sections" on page 26

"Modifying ECD Files" on page 37
"Modifying Source Files" on page 51
"Building the Binary Image" on page 54
"Installing the Package" on page 56
"Distributing Packages as LPF Files" on page 57
"Using the Embedix Tool Chains" on page 58

Note: We expect most users will appreciate the automation that Package Editor offers, but recommend that you read "Manual Method of Packaging" on page 125 at least once for an overview on how to create a package on your own. This should help you better understand the underlying Package Editor functionality.

What is an Embedix Package?

An Embedix package consists of at least three files:

- An LBC (Lineo Build Control) file containing the build control instructions for a package
- An ECD (Embedix Component Description) file containing information relevant to Target Wizard
- A source file (which could be a tarfile, SRPM file, or CVS directory) containing the source code and spec file of the package

Note: A package is permitted to have one or more patch files that will be applied after the SRPM or tarfile has been unpacked and prepared. These patches are activated by the presence of the %patches section in the LBC file.

Package Editor was designed to allow you to configure individual applications in its interface. It provides summaries (such as inheritances and dependencies) of your project's LBC, ECD, and source file entries.

Starting Package Editor

In this product, Package Editor can be invoked from either the Target Wizard menu bar or the command line—your choice.

From the Menu Bar

In the SDK version 2.4 for Windows or Linux, the option to invoke Package Editor from Target Wizard appears in the Tools menu. You may also invoke Package Editor on a package by right-clicking on a component node in the Target Wizard tree view and choosing Package Editor from the context-sensitive menu.

To invoke Package Editor from the menu bar, choose Tools > Package Editor. To open a specific package in Package Editor, choose Tools > Package Editor > File > Open Package and choose an SDK package from the drop-down list (which contains all of the packages available in the Target Wizard project you have opened).

From the Command Line

When Package Editor is installed on a linux machine, a file called packager is place in /usr/local/bin. This file is a wrapper and is marked as executable so for most linux machines, simply entering **packager** on the command line will start Package Editor. Whenever Package Editor is started, it needs to have a project that it will be working in. This must be specified on the command line with the -d option. For example, in order to run Package Editor on an 8260 project, you would pass the project's directory on the command line, like this:

packager -d /home/<*username*>/project/8260

Package Editor will always record the last project that was used so the next time you start it, you need not specify the project again unless you want to change it. In addition, you may optionally specify the package to work with on the command line by using the -p option.

Once Package Editor is up, you may also select it graphically. The name passed with the -p option should be the name of the ecd for that package. Usually this is just the name of the package itself (as it appears in Target Wizard). For example, to work on the bash

package, you would invoke Package Editor in a manner similar to the following example:

```
packager -d /home/<username>/project/8260 -p
   bash
```

As with the project, Package Editor remembers the last package that was opened and opens it automatically each time Package Editor runs unless the package is overriden on the command line.

When using one or more command-line options, use the following syntax:

```
packager [-d <project directory>] [-p <package name>]
```

Tip: You can view the usage for Package Editor by giving a -h or --help option on the command line.

For a list of command-line options, see the following table.

Table 2-1. Packager Command-line Options

Option	Description
`-d` <*project directory*>	This is the project directory that you would like to use. If you do not specify this then the packager will look at your current working directory to see if you are in a project tree. If so, it will use that project's directory. If you are not in a project tree then it will use the last project that you opened with Package Editor. If all of these methods fail then the packager will show this information and quit.
`-p` <*package name*>	This specifies the package to work on. If it is not specified then the packager will open the last package that you used in the determined project. If there was no last package open for the project then it will start with no package open.
`--ip` <*ip address*>	If Package Editor is started from windows, the ip address of the vmware/linux session must be given so that it can connect for building packages. This may be a dotted quad like 172.27.170.5 or a host name like cbuild.eng.lineo.com

Table 2-1. Packager Command-line Options

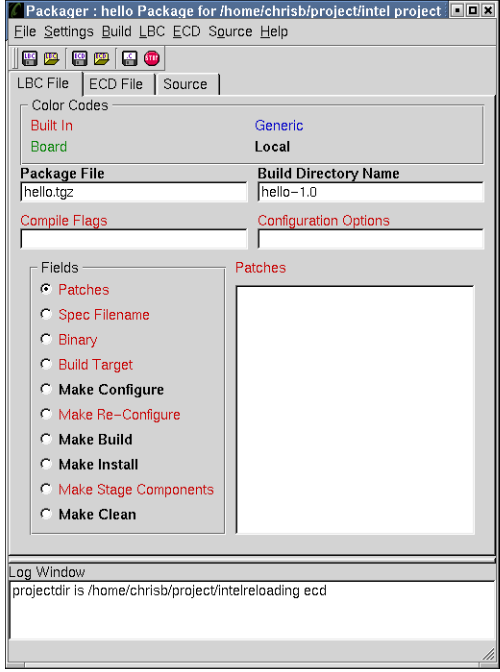
Option	Description
`--drives` <*home drive*>:<*opt/Embedix drive*>	If Package Editor is started from windows, the names of the two drives that are samba mounted need to be passed in. The first drive is the samba mount that corresponds to /home/vkit and the second corresponds to opt/Embedix. Normally this should look like: `--drives p:q`

Exploring the Interface

If Package Editor is started with no package selected, then all of the widgets are disabled.

If you specify a package on the command line or open a package within Package Editor, then the package information will be displayed and the widgets will be enabled, such as in Figure 2-1.

Figure 2-1



The Package Editor interface has four key areas:

- Menus
- Short Cut Icons
- Tabs
- Log Window

Notice that the active tab (such as LBC File in Figure 2-1) consumes the bulk of the screen real estate.

Menus

The menu bar contains the menus listed in the following table.

Table 2-2. Package Editor Menus

Menu	Description
File	File options, like "New"
Settings	Preference options, like "Editor"
Build	Build options
LBC	LBC File options
ECD	ECD File options
Source	Source options
Help	Standard help options

Shortcut Icons

Several shortcut icons have been included for your convenience.

Table 2-3. Shortcut Icons

Icon	Description
LBC	Saves the LBC file
LBC	Re-opens the LBC file
ECD	Saves the ECD file
ECD	Re-opens the ECD file
.C	Copies the opened file
STOP	Stops the current operation

Tabs

The tabs on the main interface are LBC File, ECD File, and Source. These provide text-entry areas for the properties and settings of three types of package files: LBC, ECD, and Source.

For information on the fields on these tabs, see the following sections:

"LBC File Tab" on page 27
"ECD File Tab" on page 38
"Source Tab" on page 51

Log Window

The Log Window provides a display area for build activity messages and error messages.

Overview to Creating a Package

Before explaining all of the advanced features available in the packager, we'll present two brief tutorials on automatic packaging. The package editor is intended to be used for two main purposes.

The first is to make an easy task out of modifying an existing packages. If a developer creates a BSP (Board Support Package) for the SDK, then he might need to make changes to packages for the new architecture. The package editor makes it easy to tweak existing packages until they work for a new architecture, a newer version of a package's source code, etc.

The second purpose of the package editor is to facilitate the creation of new packages (automatic packaging). The best way to leverage the toolchains available in the Embedix SDK is by creating an SDK package from existing source code. Creating a package by hand is not an easy task. It requires an understanding of LBC and ECD files and it requires a developer to have knowledge of the directory structure of projects. The package editor solves the problem by reducing and in some cases eliminating the need to learn about all of these things.

As an example of how easy creating a package can be, you are presented a tutorial for the classic “hello world” example. Note that while Package Editor does eliminate tedious work required to create LBC and ECD files, we recommend that you look at the sections describing these files for a basic understanding of what they do and how they work.

Here are the general packaging steps (with cross-references to the relevant documentation) and a tutorial.

General Packaging Steps

1. **Prepare:**

   “Downloading the Source” on page 25
   “Unpacking the Source” on page 25

2. **Configure:**

   “Modifying LBC Sections” on page 26
   “Modifying ECD Files” on page 37
   “Modifying Source Files” on page 51

3. **Make:**

   “Building the Binary Image” on page 54

4. **Install:**

   “Installing the Package” on page 56
   “Distributing Packages as LPF Files” on page 57

Tutorial: Creating a New Package

Normally when a developer wants to get an application working on an embedded system, he first develops it on his own workstation (usually an x86 machine) until it is working, and then ports it by building it with the target toolchain. We will follow this model in creating a package from scratch.

Here is a “Hello World” example. Note that even though the package does eliminate tedious work required to create LBC and ECD files, we recommend you review to section “Manual Method of Packaging”

on page 125 for a basic understanding of purpose of these files and how they work.

Normally when a developer wants to get an application working on an embedded system, the developer first develops it on his or her own workstation (usually an x86 machine) until it is working, and then ports it by building it with the target toolchain. For practice, follow this same model in creating a package from scratch.

Exercise: Create a “Hello World” Package

1. Create a directory somewhere on your machine and call it hello-1.0. (There's nothing magic about the name here, it just needs to be in it's own directory.)
2. In that directory, create a text file called hello.c and place the following content in it.

```
#include <stdio.h>
int main(void)
{
 printf("hello world\n");
 return 0;
}
```

3. Create a makefile for your hello world program. For Package Editor to automatically create a package from source, the makefile must have three key features.
   - **The makefile must build the program when called simply with “make.”** This is the normal convention for makefiles.
   - **The makefile must have a clean target.** That is, you should be able to do a “make clean” and it will clean out the directory of all built files (like .o's and programs) Again, this is normal.
   - **The makefile must have an install target** and **it must support a prefix argument.** You should be able to type make install and have the program install on the target machine. You should be able to enter:
     ```
     make install prefix=/tmp
     ```
     to have it install using the tmp directory as the root. (This concept is illustrated in the following example.)

Here is a makefile that contains two out of three key features. It has a normal install target, but does not support a prefix argument.

```
hello: hello.c Makefile
 gcc $(OPT_FLAGS) hello.c -o hello

clean:
 rm hello

install: hello
 install -d /usr/bin
 install hello /usr/bin
```

This makefile will do three things:

- ▷ Build the program if called with no arguments.
- ▷ Clean the directory of binaries if called with clean.
- ▷ Copy the program into the filesystem if called with install.

Here is the same makefile with the prefix option added.

```
prefix=/

hello: hello.c Makefile
 gcc $(OPT_FLAGS) hello.c -o hello

clean:
 rm hello

install: hello
 install -d $(prefix)/usr/bin
 install hello $(prefix)/usr/bin
```

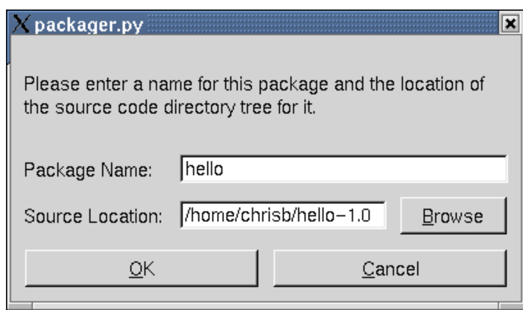
If you do a “make install,” at this point, then the prefix will be set to / and hello will be installed into a normal place. But if you do a “make install prefix=/tmp,” then hello will be installed into /tmp/usr/bin. Making this type of a modification to a makefile is usually a very simple procedure.

4. Now that you have the source file and the makefile, try testing a build, install, and clean by hand. If everything works then you're ready to turn it into a package.
5. Create a package using Package Editor.

5a. Start Package Editor by invoking it from the command line or selecting it in the File menu of Target Wizard.

5b. From the Package Editor's File menu, choose Import Source as Package. A text entry box appears prompting you for the name of the package and the location of the source code.

Figure 2-2. Import Source as Package



5c. Enter a package name and a source location (as in Figure 2-2) and then click OK.

Package Name: The name should not include spaces and should be all lower case letters.

Source Location: For the source location, enter the directory path to where the source files reside. You can click on the Browse button to look through your filesystem.

If no error message appears after a few seconds, then a Target Wizard package has been created. Click on the LBC, ECD, and Source tabs to view the files created for this package.

In order to actually see the source files, from the Source menu, choose Prepare Source for Work and then view the source directory layout in the source tab. (For a detailed explanation of each of these tabs and their functions, refer to the tab-specific modification instructions later in this chapter.

6. Do a test build in Target Wizard.

To do this, complete the following steps or refer to “Building the Binary Image” on page 54, which describes how to do a test build from within Package Editor.

6a. From the Target Wizard menu bar, choose Project > Open Recent. At the top of the list is the project that is currently opened.

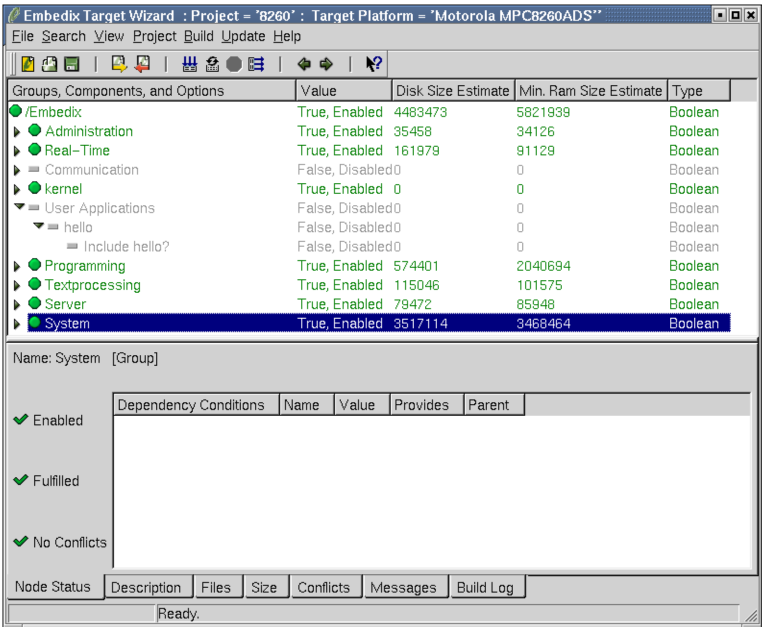
6b. Select the current project to have Target Wizard reload all of it's data. This operation may take up to a minute, so Target Wizard is not really frozen, even though it might not respond for a little while. When Target Wizard is finished reloading the project, a new main node should appear.

6c. Expand the main node to see the hello world package that was just created (such as Figure 2-3).

6d. Enable the nodes and then right-click on the hello node and select it to do a Forced Rebuild. Because this is a very small program it should build quite fast. You can view the output of the build by clicking the Log tab in the bottom portion of Target Wizard.

If everything went well and there were no errors, then the hello world program has just been built with the toolchains for the project (which was 8260 in the case of our sample project in Figure 2-3 on page 18). You can now deploy to a target and the hello program will be placed in /usr/bin (because that is where the `make install` placed it using some prefix magic).

Figure 2-3. Target Wizard Project



Congratulations. You have just created an Embedix SDK package that can be used for any toolchain or project. Creating packages this way allows developers to leverage the toolchains available in the SDK. Not all packages will be created quite so easily, but this is a good start.

If you encountered problems when Target Wizard tried to build the package or if you want multiple options for a program, then use Package Editor to fine tune the package until everything works fine.

This concludes the Hello World tutorial, which was focused on creating a new package. The above tutorial is most useful for applications designed in-house for a specific architecture or board.

The next tutorial will focus on using the package editor to create a package for an application from an external source. This can apply to packages created from Package Editor, like "hello" from the example, or pre-existing packages (like ash or flex).

Tutorial: Creating a Package for an Application from an External Source (like Open Source Applications)

While the abilitiy to create a new package is desirable and usually necessary, more often than not in Linux system development, you will obtain a piece of software from the open-source community to include on your board. Fortunately, because of the configuration mechanisms found in most open-source software, this is not a difficult task.

Exercise: Convert "Gnu Make" into an Embedix Package

The following is an example of how to convert GNU Make (a popular Linux development tool) into an Embedix SDK package for use in embedded systems.

1. "Grab" the source. You can obtain make-3.79.tar.gz from a mirror of GNU's software FTP site.
2. Because Package Editor expects to import a source tree, you can simply untar the make source tarball in your home directory by using the following command:

   `tar zxf make-3.79.tar.gz`

   A directory named make-3.79 is created in your home directory.
3. Start Package Editor, giving it the appropriate path to the desired project directory:

   `packager -d /home/`*<username>*`/project/mips`
4. From the File menu, choose Import Source as Package.
5. In the dialog box that appears, enter **`make`** for the name, enter the path to the source directory (for example, /home/ *<username>*/make-3.79), and then click OK. A minimal package is created for **`make`**.
6. Because most open source software supports **`make`**, **`make clean`**, and **`make install prefix=<`*directory*`>`**, no changes to the source are needed to get **`make`** to compile. To test this, from the Build menu, choose Force Complete Rebuild. The package should build without error. If, for some reason, there is

an error, it is displayed in the log window at the bottom of the screen. If **make** successfully built, then a tarball of the resulting binaries has been automatically created.

7. To ensure that the correct files from the tarball you built are put on the target when deployed, edit the keeplist.

7a. Select the ECD File tab.

7b. In the tree view, right-click on the node titled ''Include make?''.

7c. From the context-sensitive menu, choose Create Keeplist. This brings up a dialog with a list of files. These are the files that were built for this package.

7d. To allow the users of this package to install just **make** and not any documentation, in the list, select /bin/make.

7e. Add nodes for including the man page and the info pages as well: (1) Right-click on the `make` node and choose "Add a child to this node." You will be prompted for the name of the node. (2) Enter ''include_make_man'' and then click OK. (3) In the prompt field for this node, enter "Include man page?" (4) Edit the keeplist for this node as you did for the other node, but this time select only the file "/man/man.1/make.1." (5) Add another node to the "Make" node and name it "include_make_info." (6) Enter "Include info pages?" as the prompt and select all of the info files for the keeplist.

You now have a working ecd file, but the size information is inaccurate. If you click on the Size tab for the ecd nodes, you'll see that everything is set to 0.

8. To have Package Editor automatically compute the true size, right-click on each of the three leaf nodes (the only nodes that include a keeplist) and then choose "Compute Size Information from Keeplist." After selecting that option for each node, Package Editor calculates the size and fills in those fields.

9. Run **ldd** on the "make" on your system. You will find that "make" needs "libutil.so.1" to run.

10. Ensure that “libutil.so.1” ends up on the target along with “make” or make won't run:

10a. In the tree view of the ECD File tab, select the “Include make?” node.

10b. With the Node subtab selected, from the Attributes radio button list, choose **requires**. The word “requires” now appears over the large text-entry box on the right.

10c. In the “requires” text-entry box, type: `libutil.so.1` This indicates that make needs that file to work correctly. Now this dependency will show up in Target Wizard.

At this point everything required is done.

11. (Optional) Add descriptions to the help fields of the nodes:

11a. In the tree view, select a node.

11b. With the Node subtab selected, from the Attributes radio button list, choose **help**. The word “help” now appears over the large text-entry box on the right.

11c. In the “help” text-entry box, type the help content that you want displayed for the selected node.

12. Save all of the files and open this project again in Target Wizard. Make should show up in the ECD tree and you should be able to select it and deploy it to the target.

Tutorial: Creating Binary Packages to Use in Target Wizard

There may be times when you have a pre-built binary package that you would like to include in the target. The process of creating a package for this binary is simple using the Package Editor. The steps required are as follows:

1. Create a directory containing the binary and all needed files for it to work correctly.
2. Import the source as a package using the Package Editor.
3. Modify the “Make Build” and “Make Install” sections of the lbc tab.

4. Modify the keeplist in the ecd tab.

No build step is required. Once you have done these four steps, you have a working package for the architecture that the binary was built for. Of course, you'll need a different binary for each architecture, so this type of package won't be as portable as a source package. But in many cases where only one architecture is needed, this works well.

Below is a step by step detailed description of how to create a “hello world” binary package.

Exercise: Create a Binary Hello World

1. Create a file called hello.c with the following as its contents:

```
#include <stdio.h>
main()
{
        printf("Hello World!\n");
}
```

2. Compile the executable statically to have no library dependencies. To compile it statically, execute the command:

```
gcc hello.c -o hello --static
```

   Of course this creates a binary for an x86 architecture so if you would like to build an executable for a different architecture you will need to use the appropriate cross-compiler.

   The Embedix SDK provides cross-compiler stationary for use with the CodeWarrior IDE that makes this a simple process.

3. Once the executable is created, create a directory to house the contents of everything needed. For this hello world app, no additional files are needed, so just create a directory and put it in there.

```
mkdir /tmp/hello
cp hello /tmp/hello
```

4. Create the package:

   4a. Start the Package Editor.

4b. From the File menu, choose “Import source as a package”.

4c. For the name, enter hello (or, enter a one or two word name of the package substituting underscores for spaces). It’s best to use one word when possible.

4d. For the source location enter /tmp/hello. The Package Editor now creates a minimal lbc file and ecd file, and then “tars up” the contents of /tmp/hello. (At this point the /tmp/hello directory can be removed because it is no longer needed.)

5. Modify some lbc entries.

Package Editor makes assumptions about the existence of a makefile. In this case there is no makefile. First, change the entry in the “Make Build” section of the lbc tab by changing it from “make” to something like “`echo "nothing to build"`”. This is required because the “Make Build” needs to have something in it. There is no makefile, so we just insert a dummy command.

6. Do the same thing for the “Make Clean” section. In that section just enter something like “`echo "nothing to clean"`”.

7. The only section that needs real content in it is the “Make Install” section. In this section enter in the commands needed to place the binary and supporting files into the correct location in the target filesystem. As with the other packages, the “Make Install” section of this package will need to reference the $BUILD_ROOT variable as the virtual root of the target filesystem. This means that if we want to executable hello to end up in /usr/bin then we need to execute the command:

```
cp hello $BUILD_ROOT/usr/bin
```

instead of

```
cp hello /usr/bin
```

As with normal makefile installs, we need to also create all the directories before we use them.

Use the install command because it allows the user to specify the mode, owner, and group of a file or directory as it is created. Our "Make Install" section now look like this:

```
install -d -m 755 -o root -g root $BUILD_ROOT/usr/bin
install -m 755 -o root -g root hello $BUILD_ROOT/usr/bin
```

8. Click the lbc disk button to save the LBC file after making these modifications.
9. Now that we have modified the "Make Build", "Make Clean", and "Make Install" sections of the lbc, we can practice building by selecting "Force Complete Rebuild" from the "Build" menu. Of course no actual building will take place, but the different sections of the lbc will be executed and the binary file will be put in the proper location to be deployed to the target. If there are any problems or errors during the build, check them and modify the "Make Install" section as needed.
10. Once you are able to complete the build successfully, click on the "ECD File" tab. This will show the ecd tree for this package. You will want to modify the keeplist for the Include node. In my tree, the last node is labeled "Include hello?". If you right-click on the node and select "Create Keeplist" a dialog box will be brought up listing all of the files that were "installed" in the "Make Install" section of the lbc. Select the files that you want to be installed onto the system in this dialog and then close it.
11. If you would like to calculate the file sizes so that they are accurately represented in Target Wizard, then right-click on this same node and select, "Compute Size Information from Keeplist". Make sure that you save the ecd file after making the above modifications.
12. You can now re-open the current project in Target Wizard and the package that you created will appear under the "User Applications" node of the ecd tree. You can then enable or disable this package and Target Wizard will treat it just like any other package. As with source packages, you can modify the ecd of this package in any way you choose. More detailed information about lbc and ecd files are described elsewhere in this documents.

Downloading the Source

When you intend to create a package from open source, you may need to download the source tarball from an FTP site or other Web site.

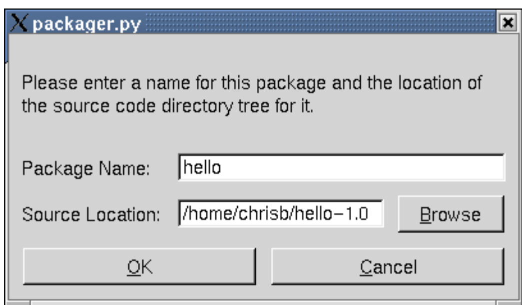
Unpacking the Source

You need to unpack (untar) the source code before doing any work on a package.

1. Untar the downloaded tarball into a directory and select "Import source as a package."

   A dialog box appears prompting for the name of the package and the location of the source code (like the sample shown in Figure 2-4).

Figure 2-4. Import Source as Package



2. Use the dialog box to enter or browse to the source directory location.

   The name should not include spaces and should be all lower case letters.

3. Click OK.

   If no error message appears, then a Target Wizard package should be created.

You can now click on the LBC, ECD, and Source tabs to view the files created for this package. In order to actually see the source files, however, you'll need to choose Prepare Source for Work from the Source menu. Then you should see the source directory layout in the source tab.

You can now begin making modifications to the LBC file (which controls how the package is built). For more information, see “Modifying LBC Sections” on page 26.

Modifying LBC Sections

Once you have unpacked the source, you can begin making modifications to the package’s LBC file, which controls how the package is built from start to finish. The related topics are:

“What is an LBC?” on page 26
“LBC File Tab” on page 27
“LBC File Sections” on page 30
“LBC File Inheritance” on page 32
“Build Variables in LBC Files” on page 33

Note: When you have finished modifying an LBC file, from the Package Editor menu bar, choose LBC > Save LBC.

Tip: Modifying the LBC sections for a package and test-building the package go hand in hand. Whenever an LBC section is modified, you should test build the package to make sure that the changes had the desired affect. For build instructions, see “Building the Binary Image” on page 54.

What is an LBC?

LBC stands for “Lineo Build Control.” An LBC file contains build instructions for a package.

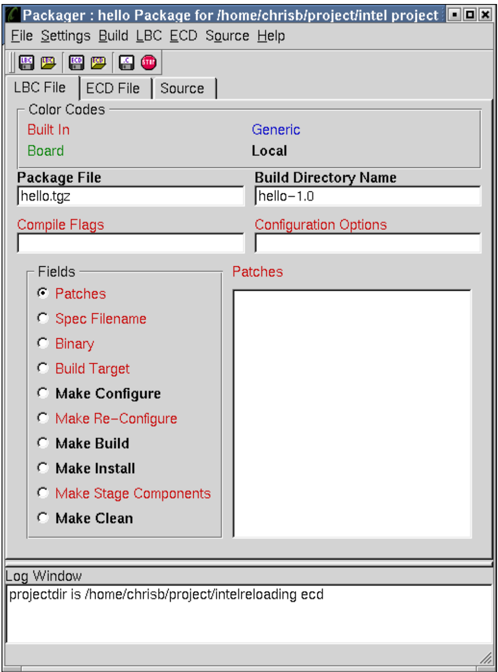
A package may get build instructions from four different sources—built-in defaults (in a file called builder.pm), a generic LBC file, a board-specific LBC file, or a local LBC file, with preference being given to the entries in the local LBC file. (This is discussed in detail in the section “LBC File Inheritance” on page 32.)

LBC File Tab

The LBC File tab is one of three main tabs in Package Editor interface and is the tab displayed by default when you start Package Editor.

When you view LBC file settings from this tab, you are actually viewing the aggregate LBC—the merged version of all applicable LBC file sections from four files. But when you edit LBC file sections here, you are essentially editing the local LBC file for a package.

Figure 2-5. LBC File Tab



The LBC File tab can be divided into three areas: color codes, four fields with small text-entry boxes, and a list of fields list (radio buttons) with a large text-entry box.

Fields

The four small text-entry boxes near the top of the LBC File tab are for LBC sections that are normally only one line long: Package File, Build Directory, Name, Compile Flags, and Configuration Options.

The radio buttons on the left side represent multi-line LBC sections that are allowed in an LBC file (such as Patches, Spec Filename, and Binary). When a radio button is selected, the contents of the corresponding LBC section is displayed in the large text-entry box on the right.

Color Coding

At the top of the LBC File tab is a color code that lists the four possible sources of an LBC section (or field) on this tab and their corresponding colors. The name of each field is colored red, blue, green, or bold black. This color code applies to the one -line fields (like Package File) and the multi-line fields (like Patches).

- **Red = Built In**: Coming from values that have default entries in the builder (which are displayed in the text-entry box). Most of these fields will be empty. But a few, like Make Configure and Make Build, do have built-in values for the builder.
- **Blue = Generic:** Coming from the LBC in the generic directory. These are fields that apply to the package for every project and board.
- **Green = Board:** Coming from the LBC in the board directory. These are sections that have been changed or added to from the generic LBC to get the package to build properly for the specific BSP that your project is.
- **Bold Black = Local:** Coming from this project's LBC file. These fields do not apply to the package in any other project or BSP.

With color coding you can see all of the LBC settings that the builder will access when it attempts to build the package. All modifications made by the user are placed in the local LBC file. (For related information, see "LBC File Inheritance" on page 32.)

If you modify a field on this tab, the heading will immediately turn bold black, indicating that this modification will go into the local LBC file. To revert to the value in the board, generic, or built-in LBC, simply erase the contents of the text-entry box; the wizard will fill in the old value automatically.

Note that when the LBC is changed, an asterisk appears next to the name on the tab. This indicates that changes have been made that have not been saved. When you save the LBC, the asterisk will disappear. In addition, if you attempt to close the project or reload the LBC, you will be asked if you would like to save the LBC before doing so.

LBC File Sections

The different sections of an LBC file specify to the builder how to complete a phase of building the package. Here is a complete list of all the fields available in an LBC and how they are used.

Table 2-4. LBC File Section Descriptions

LBC File Section	See also	Description
Package File	"%pkg_file" on page 130	This is the name of the original source file. Usually this will be a tarball of some kind.
Build Directory Name	"%bld_dir_name" on page 131	This is the build directory for the source. Usually a tarball untars into a directory (for instance, untarring bash-1.2.tgz will create the directory bash-1.2) You can have the packager make an educated guess as to what the build directory is by choosing Determine Build Directory, which is found on the LBC menu.
Compile Flags	"%cflags" on page 131	These are the CFLAGS that will be applied to the package when building.
Configuration Options	"%cfgopts" on page 131	Extra options to be used in the configuration stage (not normally used).
Patches	"%patches" on page 130	The name of any patches that need to be applied to the source after untarring it. These are applied in the order that they appear in this section.
Spec Filename	"%spec" on page 131	For SRPMs, this is the name of the spec file to be used for building (not normally used).
Binary	"%bin" on page 131	Used only for the kernel, this is the name of the binary that is produced.
Build Target	"%bld_targ" on page 131	Used only for the kernel, this is the specific name of the make target required to build the linux kernel.

Table 2-4. LBC File Section Descriptions

LBC File Section	See also	Description
Make Configure	“%makec” on page 133	This contains the instructions to configure the package. Equivalent of runnning **`./configure`** on most open source packages. In fact, most packages have **`./configure`** with some options for this step.
Make Re-Configure	“%makerc” on page 134	This contains the instructions to re-configure a package if it has already been configured and some of the options change.
Make Build	“%makeb” on page 134	This contains the instructions to actually build the package. This is the equivalent of executing make on the package. For most packages this will include the command **`make`** and maybe a few other modifications.
Make Install	“%makei” on page 134	This contains the instructions to install the package. When the builder runs this section, the package should not install onto the host system, but into a temporary staging area. The path to this staging area is contained in the environment variable $BUILD_ROOT. For a lot of packages, this step looks like: `make install prefix=$BUILD_ROOT`
Make Stage Components	“%makest” on page 135	This contains instructions to build components that are needed to build the package but that do not need to be installed. This could be used if you need to build library to build the package, but the library doesn't need to be on the target machine at run time.
Make Clean	“%makedc” on page 135	This contains the instructions to clean the source build directory. Usually this consists of **`make clean`** or something similar.

LBC File Inheritance

Default build instructions are contained in the builder.pm file and should not be modifed. They can, however, be overridden with LBC file entries. LBC files can exist in three directories of a project:

<*project*>/config-data/buildcontrol/generic
<*project*>/config-data/buildcontrol/board
<*project*>/config-data/buildcontrol/local

These directories have an inheritance mechanism built in that allows a package to be customized easily for a specific board or even project.

The **generic** directory contains package-dependent LBC files and sections that are valid for any board or project.

The **board** directory contains architecture-dependent LBC files and sections that override above behaviors and fix very specific architecure build problems.

The **local** directory contains local project development LBC files and sections that override any of the above sections. Personal software development should be specified in this section.

Sections in LBC files from each of these locations are merged, giving priority first to local, then to board, and then generic. For this reason, end-user LBC files should reside in the local directory.

Example:

The following example shows the masking behavior for a project that has LBC files with section entries in the four locations described. In this example, the sections included in each LBC file are represented with letters.

A = Make Project (%makep)
B = Make Configure (%makec)
C = Make Build (%makeb)
D = Make Install (%makei)
E = Make Clean (%makedc)

```
Location       LBC File Sections
builder.pm     A B C D E
generic        A - - D E
board          - - C - E
local          A - - - -
               | | | | E from board/<pkgname>.lbc
               | | | D from generic/<pkgname>.lbc
               | | C from board/<pkgname>.lbc
               | B from Builder.pm (SDK default)
               A from local/<pkgname>.lbc
```

The point of this example is to demonstrate that each of the files can have identical sections, but the behavior is dependant on a specific hierarchy. If you want to override a certain section, then you only need to include that section (plus the only required section, %pkg_file or Package File) in the LBC file. In other words, you do not need a complete LBC file in order to override one section.

Build Variables in LBC Files

When a package is built using Target Wizard, Target Wizard places in the environment a number of useful variables. These can be used in different sections of the LBC to customize your builds.

For example, the variable **`$BUILD_ROOT`** is almost always used in the Make Install section of an LBC file in the following way:

```
make install prefix=$BUILD_ROOT
```

Comprehensive List of Variables

Here are the available build variables and their default settings (where applicable):

- **PATH** - /opt/Embedix/tools/bin:/opt/Embedix/bin/opt/Embedix usr/bin:/opt/Embedix/usr/local/bin
- **HOST_PREFIX** - i386-linux needed by some packages (e.g flex)

- **TWTOOLS_NATIVEPREFIXED_PATH -** /opt/Embedix/tools/native-linux/bin:$PATH This is used in cross compiler aware packages, when the CC variable needs to reference the host (x86) compiler (see glibc.lbc).
- **TWTOOLS_TARGETPREFIXED_PATH -** /opt/Embedix/tools/<cross>/bin/$PATH This is used when the CC variable needs to reference the cross compiler (e.g spoofed).
- **NATIVE_PREFIX -** native-linux This can be used in lbc files to force a compiler symbol refer to the host (x86) compiler (e.g. modutils sets HOSTCC=$NATIVE_PREFIX-gcc).
- **INCLUDE_PATH_DEFAULT -** <dev_image>/usr include:<dev_image>/usr/local include:<:$TC_GCC_INCLUDE_PATH This is used by the native-linux/bin/tw_wrapper spoofing script.
- **LIBRARY_PATH_DEFAULT -** <dev_image>/usr/lib:<dev_image>/usr/local/lib:<dev_image>/lib:$TC_GCC_LIBRARY_PATH This is used by the native-linux/bin/tw_wrapper spoofing script.
- **LD_LIBRARY_PATH -** This parameter is deliberately cleared.
- **TC_SIZE_SHORT, TC_SIZE_INT, TC_SIZE_LONG, TC_SIZE_FLOAT, TC_SIZE_DOUBLE -** These parameters are used by the toolchains. Their values are set in /opt/Embedix/tools/<arch>.tcconfig.
- **SOURCE_DIR -** <project>/build/rpmdir/SOURCES
- **BUILD_DIR -** <project>/build/rpmdir/BUILD
- **OPT_FLAGS -** This variable takes on the value of %cflags set in bsp_config. This may be overidden in the package lbc file.
- **ARCH -** This variable takes on the value of %arch set in the bsp_config file.
- **BUILD_ROOT -** <project>/tmp/<package> This variable names the transient path used to stage the root directory for the install phase of building.

- **PACKAGE_NAME** - This variable holds the current package name.
- **CFG_OPTS** - This options can be used to pass extra options to the configure stage.
- **DEV_IMAGE** - <project>/build/dev_image This variable point to the location where packages install files needed by other packages. For instance, glibc installs it's header files here.
- **CROSS** - This variable is set to the value of the token %cross in the bsp_config file (e.g powerpc-linux).
- **BUILD_TARG** - This variable reflects the value of %bld_targ in the bsp_config file.
- **BINOUT** - This variable reflects the value of %bin in the lbc file.
- **RPMDIR** - <project>/build/rpmdir
- **LINUX_HDR_DIR** - This variable points to the location of the linux header files.
- **TARDIR** - <project>/build/tarfiles The final output of a complete build for a package is a tarfile, this file is place in this directory under the name $TARDIR/<package_name>.tar.gz.
- **PKG** - This variable holds the current package name (same as PACKAGE_NAME).
- **CVSROOT** - This is for packages that get there source from cvs. This variable specifies the cvsroot.

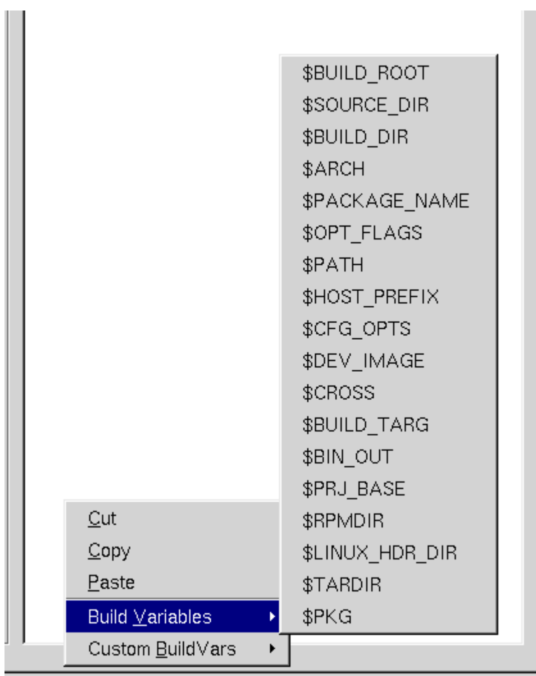
Using Common Variables

To view or use a list of commonly used build variables for any multi-lined LBC section (or field):

1. On the left side of the LBC tab, choose the appropriate radio button. The contents of the corresponding LBC section appear in the large text-entry box on the right.
2. Place the cursor in the large text-entry box where you would like to insert a variable.

3. Right-click and then choose Build Variables from the context-sensitive menu . A list of build variables appears.
4. Select a build variable to insert it at the location of the cursor.

Figure 2-6. Build Variables



In addition to built-in build variables, it is possible to use ECD buildvars in LBC files. This allows a package to be built differently depending on the ECD options that are enabled or disabled for that package in Target Wizard. For information on creating BuildVars in ECD files, see the ECD section of this document.

To use an ECD BuildVar in an LBC section:

1. Select the desired section from the radio button group.
2. Place the cursor in the desired location in the text box and right-click.
3. From the right-click menu, select custom buildvars. A list of all buildvars available from the ECD appear.

4. Select the buildvar that you want to insert and it will be placed at the location of the cursor.

It's important to note that there is nothing magical about the way that Package Editor inserts the variable names. You could type a variable in (such as $BUILD_ROOT) and get the same result as when you select it from the menu. The menus are there for your convenience to help you avoid some typing or having to look up valid variables.

Modifying ECD Files

Tip: If you have modified your LBC file, be sure to test your changes by doing a test-build before proceeding. For build instructions, see “Building the Binary Image” on page 54.

ECDs are used to allow multiple configurations for an Embedix package. After you get the package building correctly, you can modify the ECD so that the user can select options for the package.

This section covers the following topics:

“What is an ECD?” on page 37
“ECD File Tab” on page 38
  “ECD Nodes in the Tree View” on page 38
  “ECD Node Properties in Subtabs” on page 40
  “ECD Node Properties in Attributes List” on page 43
“Build Variables in ECD Files” on page 48

Note: When you have finished modifying an ECD file, from the Package Editor menu bar, choose ECD > Save ECD.

What is an ECD?

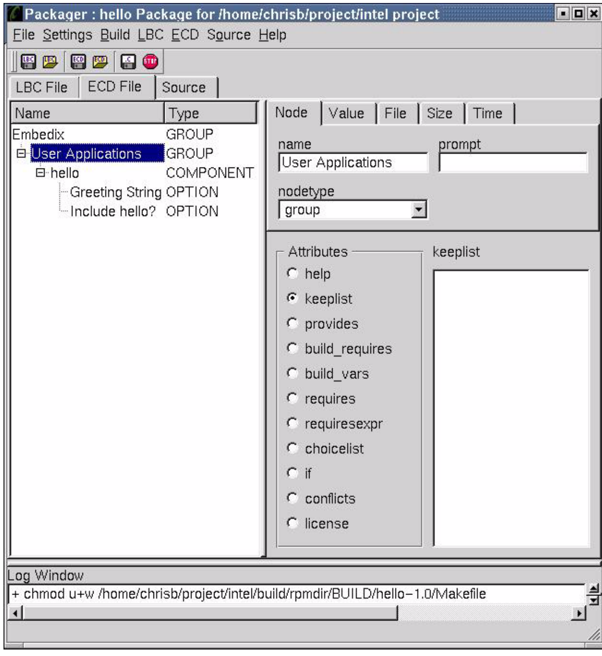
ECD stands for Embedix Configuration Description. This is an XML-like file that describes different configuration options for a package.

This includes things like the help for a given node, the keeplist for options, etc.

ECD File Tab

ECD files have many properties—all of which can be edited using Package Editor and the ECD File tab.

Figure 2-7. ECD File Tab



The ECD File tab (as shown in Figure 2-7) can be divided into two sections:

- ECD nodes (Tree View)
- ECD node properties (subtabs with small text-entry boxes and the Attributes list with radio buttons and a large text-entry box)

ECD Nodes in the Tree View

On the left side of the tab is a tree view of all of the ECDs for the current project. This is much like the tree view in Target Wizard,

except that there is no dependency checking and Package Editor shows autovars (which are not visible in Target Wizard).

When you click on a node in the tree view, its information displays under the subtabs and attribute fields on the right (where you can modify the node's information as needed).

You can expand and collapse a portion of the tree by clicking on the + or - boxes next to nodes that have children.

Types of Nodes

In the tree view, each ECD node is listed as one of four main types:

- **Group** nodes are the top-level nodes. These correspond to categories of packages. Group nodes can be nested and their children are always component nodes or other group nodes. The root node of every project is the Embedix group node.
- **Component** nodes describe packages. There is a one to one correspondence between component nodes and packages in the project. Enabling a component node enables a package and vice versa. Component nodes can only have group nodes as their parents and autovars and option nodes as their children.
- **Option** nodes describe package configuration options. These include things like how to build the package, which files from the package end up on the target, etc. Option nodes can have option nodes as children and option nodes and component nodes as parents.
- **Autovar** nodes act similar to option nodes in that they provide configuration information to a package. The difference is that the values of autovar nodes are calculated by Target Wizard, not set by the user. Autovar nodes are the only nodes that do not show up in the tree view in Target Wizard.

These node types are also discussed in “Why and How to Use ECD” on page 44.

Edit a Node

You can right-click on nodes in the tree view to perform certain operations.

Add a child to this node - Creates a new empty node as the child of the node that you right clicked on. You can then modify the information for this child node.

Cut - Cuts the selected node (and all of its children) and places it in the packager's internal clipboard for pasting later.

Copy - Makes a copy of the current node (and all of its children) in the packager's internal clipboard for pasting later.

Paste - If there is a node in the clipboard, it pastes that node (and children) as a child of the selected node.

Write ECD for this node to the log window - Writes the ECD information for this node (and children) to the log window for inspection.

Create Keeplist - Brings up a dialog box that allows easy keeplist creation. This option will only work if the package has already been built and a tarball exists for it.

Warning: If you copy a node or set of nodes and then paste them somewhere else, you will need to rename them. Otherwise you may have more than one node in the ECD with the same name and Target Wizard will not function correctly.

ECD Node Properties in Subtabs

On the upper-right side of the ECD File tab are the subtabs: Node, Value, File, Size, and Time, which are collections of one-line project properties for the node that is currently selected in the tree view. The subtabs and their properties are outlined in the following tables.

Table 2-5. Node tab

Node tab	*Contains values that every node should have.*
name	Every node must have a name and that name must be unique to the entire project tree. This name should not include spaces and by convention contains only lower case letters and underscores.
prompt	The text that appears for the node in the tree-view for Target Wizard. If this in not set, then the name is used instead. This can contain any text desired and should describe the node in a few words.
nodetype	This specifies what type the current node is. The available types are group, component, option, and autovar.

Table 2-6. Value tab

Value tab	*Describes the value for a node. If nothing is specified here for a node then it is assumed that the node is a boolean (can be enabled and disabled) and is by default, disabled.*
type	Describes the value type for this node. The available types are: ▶ **int:** Indicates that the value of the node is an integer. This allows the user to set the value to an integer in Target Wizard. ▶ **int.oct:** Indicates that the value of the node is an octal integer. Otherwise, same as above. ▶ **int.hex:** Same as above except that the value is in hexadecimal instead of decimal. ▶ **string:** Indicates that the user can set the value of the node to a string. ▶ **bool:** Indicates that the node has two states, on and off. By default, all nodes are bools unless specified otherwise. ▶ **tristate:** Special type used only for the kernel. Allows a node to be enabled, disabled, or set to module.
default_value	The initial value of the node. The node will keep this value until a user modifies it.
range	Specifies the range if numbers that are allowed. Obviously this only applies to integer types. Examples of entries for this field are 0:100 or 0x20:0x4f.

Table 2-6. Value tab

value	<Advanced use only>

Table 2-7. File tab

File tab	*Describes some file information associated with the node.*
filename	<Deprecated, but kept for backwards capability.> Users should not modify this.
srpm	For packages that use SRPMs only.Indicates the name of the srpm to use.
specpatch	For packages that use SRPMs only. Indicates the name of the patch to apply to the specfile of the SRPM before building the package.

Table 2-8. Size tab

Size tab	*Contains size information. This information is used in Target Wizard to calculate the needed storage and ram size for the target board.*
storage_size	The size, in bytes, needed to store the files that are included in this node. This should only be set in nodes that have a keeplist.
static_size	The static size of all executable files for the node. It indicates how much ram is needed on the target to run the executables in this package.
min_ dynamic_size	The minimum amount of dynamic memory that the files in this node will need in order to run.
transient	Indicates if the program runs continually or just periodically on the target machine. The value should be specified as 1 or 0. If transient = 1 then the RAM size values are not aggregated into the sum that is used to check the target RAM requirements.

Table 2-9. Time tab

Time tab	*Contains values for times associated with the files in the current node.*
build_time	If the package takes a large amount of time to build, a rough estimate of the time needed to build the package may be given here. Time is measured in seconds.
startup_time	If the executables for the node take a long time to start up on the target, a rough estimate of the time needed to startup may be specified here. Time is measured in seconds.

ECD Node Properties in Attributes List

The lower-right side of the ECD File tab contains the Attributes radio buttons and a large text-entry box. Like the subtab fields, these attributes are also properties of the node that is currently selected in the tree view. But, because these fields are typically multi-line entries, a larger text-entry box is provided.

Click on an attribute's radio button to view its current setting in the large text-entry box. The title over the text-entry box also indicates which attribute (or ECD field) is being displayed.

The bolded items in the Attribute list are those that contain text. The non-bold (or regular) items in the list are those that are empty.

Here are descriptions of each of the attributes or multi-line fields.

- **`help`** - The help that will be displayed in the description tab of Target Wizard when this node is selected. This should include a description of what the node is and when it should be enabled or disabled.
- **`keeplist`** - Contains a list of files that will be placed on the target machine if the node is enabled.
- **`provides`** - Contains a list of the services that the node provides. This is used in dependency checking. Each line should be one word.

- **`build_requires`** - Contains a list of the packages that need to be built before the current package is built. For example, a package, foo, that uses the ncurses library would need ncurses to be built before it was built. By including ncurses in this section, Target Wizard would ensure that ncurses was built before foo.
- **`build_vars`** - Contains a list of environmental variables that will be exported to the build environment before the package is built. These expressions can be simple, such as LINK_TYPE="static" or more complex. The buildvars will only be exported if the node is enabled.
- **`requires`** - Contains a list of the packages/services that are required for this node. This could include things like ncurses, perl, or ash.
- **`requiresexpr`** - Contains an expression describing the requirements for this package. This could be something like (bash||ash) or (CONFIG_SOUND=='y') && (CONFIG_SOUND_OSS != 'n' || CONFIG_ALSA != 'n'). This allows requirements to be more versatile and complex.
- **`choicelist`** - Allows a user to select a value for the node from a number of choices. Each line contains two items. The first item is the string to be displayed and the second item is the actual value that the node will take on. Here is an example.

```
100 100
50 50
25 25
10 10
```

This will display a choice for the user to choose either 100, 50, 25, or 10 and the nodes value will be set to that number accordingly.

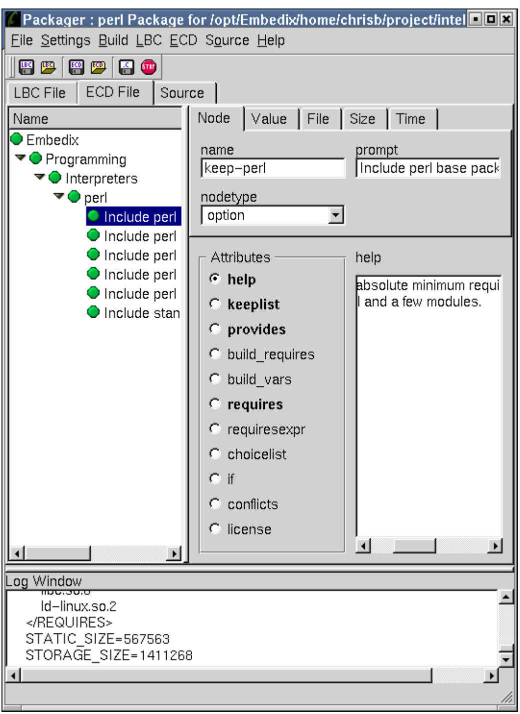
Why and How to Use ECD

Inside of Target Wizard, all of the ECDs for a given project are combined to form one tree. The top level nodes are group nodes, below groups are component nodes which represent the packages themselves and under most packages are a number of option nodes.

These option nodes allow a user to choose different configurations for a package.

For example if you create an i386 project and then navigate down the tree to Programming > Interpreters > perl, you will see a number of options below the perl node (see Figure 2-8).

Figure 2-8



All of these options are binary (meaning they are either on or off, other types of options take numerical values, etc.) and determine which files are included in the package. You can choose to include or not include the pod utilites, documentation, examples, etc. and depending on what options are chosen, different files will be put on the target. Each of these nodes represents an ECD node. Since these options affect which files are kept from the build, they use the keeplist attribute of the ecd node.

Other nodes in other packages can affect which libraries are linked to an executable, what options are passed to the ./configure script for that package, or whether it is built statically or with dynamic libraries. For most packages designed in-house by companies using the SDK for their own products, not a lot of options will be needed because the developers know exactly how the package should be built and used. Packages intended for a large audience, however, should give the user a number of options so that the exact desired configuration can be achieved.

Option nodes typically have the following attributes. Each attribute is capitalized. Usually the TYPE is a bool, meaning that it can be either enabled or disabled. The NAME of option nodes should include numbers, letters, and underscores, but no spaces. A brief description of the node should be put in the PROMPT field such as "Include Documentation?" or "Build Statically?". In addition, a longer explanation of the option should be placed in the HELP section.

If the option affects which files are placed on the target, then the option will contain values in the KEEPLIST field. If there are files in the KEEPLIST, then both the STATIC_SIZE and STORAGE_SIZE should be filled in as well so that Target Wizard can accurately determine needed disk and memory size requirements. (These last two items can be filled in automatically by the Package Editor.)

If the package needs external libraries or services (such as libc or inetd), then these should be placed in the REQUIRES section of the ECD node. If the option affects the build in some way then it will most likely have some values in the BUILD_VARS section.

Here is an example of the ''Include perl base package?'' node for the perl package. This is the actual ECD file which has an xml-like syntax so it should be easy to tell which values apply to which fields.

```
<OPTION keep-perl>
    TYPE=bool
    DEFAULT_VALUE=1
    PROMPT=Include perl base package?
    <HELP>
       This is the absolute minimum required to use perl. It provides /usr/bin/
       perl and a few modules.
    </HELP>
    <KEEPLIST>
        /usr/bin/perl
        /usr/bin/perl5.00502
        /usr/lib/perl5/5.00502/AutoLoader.pm
        /usr/lib/perl5/5.00502/Carp.pm
        /usr/lib/perl5/5.00502/Exporter.pm
        /usr/lib/perl5/5.00502/IO
        /usr/lib/perl5/5.00502/Thread
        /usr/lib/perl5/5.00502/Tie/Array.pm
        /usr/lib/perl5/5.00502/Tie/Handle.pm
        /usr/lib/perl5/5.00502/Tie/Hash.pm
        /usr/lib/perl5/5.00502/Tie/RefHash.pm
        /usr/lib/perl5/5.00502/Tie/Scalar.pm
        /usr/lib/perl5/5.00502/Tie/SubstrHash.pm
         /usr/lib/perl5/5.00502/i386-linux-thread/Config.pm
         /usr/lib/perl5/5.00502/i386-linux-thread/auto/Errno
          /usr/lib/perl5/5.00502/i386-linux-thread/auto/GDBM_File/GDBM_File.bs
           /usr/lib/perl5/5.00502/i386-linux-thread/auto/GDBM_File/GDBM_File.so
           /usr/lib/perl5/5.00502/i386-linux-thread/auto/GDBM_File/autosplit.ix
        /usr/lib/perl5/5.00502/i386-linux-thread/auto/Getopt/Long
        /usr/lib/perl5/5.00502/strict.pm
        /usr/lib/perl5/5.00502/vars.pm
        /usr/lib/perl5/site-perl/i386-linux-thread
    </KEEPLIST>
    PROVIDES=/usr/bin/perl
    <REQUIRES>
        libdl.so.2
        libpthread.so.0
        libc.so.6
        ld-linux.so.2
    </REQUIRES>
    STATIC_SIZE=567563
    STORAGE_SIZE=1411268
</OPTION>
```

The first thing to note is that the options name is ''keep-perl''. This defines the name of the node and can be used in REQUIRES sections of other nodes. For instance, the ''Include perl examples?''

node has keep-perl in it's REQUIRES sections because the examples need the perl base package in order to run. Next is the TYPE which for this node (and most option nodes) is bool. The DEFAULT_VALUE defines what the node will be set to initially before the user touches anything.

Use 1 for on or 0 for off. The HELP section is what is displayed in the description tab at the bottom of Target Wizard when the node is displayed. The next section (and by far the longest) is the KEEPLIST section. This lists all of the files that will be placed on the target if this node is enabled. Each of these files must be created when the package is built or must already exist in the source tarball or srpm. The KEEPLIST can contain actual files or directories. If a directory is specified then all of the contents of that directory will be placed on the target (include subdirectories whithin that directory).

The next section is the PROVIDES section. This is the complement of the REQUIRES section and is used for dependency checking. If another package needs perl to run then it can include /usr/bin/perl in it's REQUIRES section and it's node will be unfulfilled until a node that provides /usr/bin/perl is enabled. The REQUIRES section for this node lists all of the libraries that are needed to run perl. Nodes that provide these files will need to be enabled in Target Wizard or this node will show up as unfulfilled as well. The dependency information for a node will show up in the Node Status tab of Target Wizard. The last two fields, STATIC_SIZE and STORAGE_SIZE are the amount of memory needed to run and the amount of disk space required for this package respectively.

Build Variables in ECD Files

In addition to KEEPLIST fields, the other common way for an option to affect a package is by using BUILD_VARS fields.

This can be accomplished in one of two ways. The first and easier way is to include a number of name value pairs in the BUILD_VARS section. Here is a simple example:

```
FOO="BAR"
LINK_TYPE="static"
AGE=12
```

If this is in the BUILD_VARS section of an option, then when that option is enabled, the variables FOO, LINK_TYPE, and AGE will be created as environmental variabls with the associated values when a build occurs on that package. These variables can then be referenced in sections of the LBC file for that package or even directly in makefiles or source files. Here is a slightly more useful example. Suppose that you're creating a perl package and you want the option to include or not include threading when it is built. You might then create an option node and have the following entry in it's BUILD_VARS section:

```
WITH_THREADS="--with-threads"
```

You can turn on or off thread support in perl by passing "--with-threads" or "--without-threads" to the ./configure script so you could include the following in the '''Make Build''' section of the LBC.

```
./configure $WITH_THREADS
```

Now when you build the perl package, if this node is enabled the builder will execute:

```
./configure --with-threads
```

Otherwise it will simple execute:

```
./configure
```

Because the option is not enabled and WITH_THREADS will not be exported. If a variable is referenced in shell and it has not been defined then it is simply replaced with the nothing by the shell. This method of using BUILD_VARS can be used in a number of circumstances to affect the build, usually in shell scripts and in the makefiles themselves.

The other way to use BUILD_VARS is to export the actual value of the node. Although most option nodes are of type bool, it also to have them of type int and string. In this case the user actually enters a value for the node in Target Wizard. When this is the case, you can use a special form of BUILD_VARS that references $VALUE which is the actual value of the node itself. Following is an example of a very simple option node that uses this.

```
<OPTION include-name>
    TYPE=string
    PROMPT=Include my name?
    HELP=Includes your name in the program.
    <BUILD_VARS>
        MY_NAME=$VALUE
    </BUILD_VARS>
</OPTION>
```

In this case, the user can right click on the include-name node and enter in a string. If I enter the string "Bob Jones" for the value then when the package builds, an environmental variable will be created with "Bob Jones" as it's value. This can also be used in conjunction with the CHOICELIST section of an ecd. Here is an example from the linux ecd.

```
<OPTION Processor_family>
          TYPE=string
          DEFAULT_VALUE=CONFIG_M686
          PROMPT=Processor family
          <CHOICELIST>
                        386     CONFIG_M386
                        486     CONFIG_M486
            586/K5/5x86/6x86/6x86MX     CONFIG_M586
            Pentium-Classic  CONFIG_M586TSC
                Pentium-MMX  CONFIG_M586MMX
            Pentium-Pro/Celeron/Pentium-II     CONFIG_M686
           Pentium-III/Celeron(Coppermine) CONFIG_MPENTIUMIII
                  Pentium-4 CONFIG_MPENTIUM4
            K6/K6-II/K6-III      CONFIG_MK6
            Athlon/Duron/K7      CONFIG_MK7
                     Crusoe  CONFIG_MCRUSOE
                 Winchip-C6 CONFIG_MWINCHIPC6
                  Winchip-2 CONFIG_MWINCHIP2
            Winchip-2A/Winchip-3 CONFIG_MWINCHIP3D
                CyrixIII/C3 CONFIG_MCYRIXIII
          </CHOICELIST>
          <BUILD_VARS>
            PROCESSOR=$VALUE
          </BUILD_VARS>
</OPTION>
```

When using a CHOICELIST, each line has two strings. The first string is what will show up in a list to the user and the second string (or number) is what the $VALUE will be set to if the user selects that item.

Modifying Source Files

You can view or edit source files from the Package Editor interface.

Source Tab

The source tab displays the source files for the current package. If the tarball or srpm for that package has been exploded out then the directory structure will appear in the source file tab. If no files appear, then you can select to “unpack sources for work” from the Source menu or “Force Unpack Sources” from the Build menu. Both of these steps unpack the source (either in srpm or tarball form) into a build directory.

The pane on the left of the Source Tab displays a tree structure that represents the source directory heirarchy for the current package. You can expand and collapse directories by double clicking on them.

The text area on the right of the source tab is a simple text editor used for making small modifications to source files. Clicking on a text file will bring the contents of the file up in the edit text area. Clicking on a directory will bring up a directory listing in the text editor. If a file is not a text file, then the text editor will be empty when that file is selected.

Using an External Text Editor

You may modify the source file in the textbox or right-click and choose to open it in your favorite text editor. The built-in editor is intended to be used for small modifications to source files and not for large edits.

To use your favorite text editor on the source files:

1. Specify the path to the editor.

1a. From the Settings menu, choose Editor and then enter the path to the text editor in the text entry box. If you do not know the actual path, you may click on the '''Browse''' button to select it from your filesystem.

1b. When done, click on OK.

2. Once the editor has been set, right-click on any file and then choose “Open this file in an editor.” This will open the text editor with the contents of the file.
3. Modify the text file using your preferred editor.
4. Once you have made your changes, from the Source menu, choose Reload Current File to reload the file in the text area.

Other Options

In addition to editing, you can also delete a file by right-clicking on a file and selecting '''Delete This File'''. You will be prompted to make sure that you want to delete the selected file. You may save a file by selecting '''Save This File''' from the right-click menu or clicking on the disk icon on the toolbar that has the .c extension on its label. If you make a modification to a file and the select another file you will be prompted to save the modified file or disregard the changes.

For makefiles, there is another option available in the right-click menu. When you right-click on a makefile, an option labeled “Go To Makefile Section” appears. Select this option and a secondary menu will appear with each of the makefile targets in it. Selecting one of those targets will move the cursor to the target's location in the makefile.

Viewing “Diffs” and Making Patches

Because Target Wizard builds packages from the source tarball or srpm, it's possible that any changes made to the source tree of a package will be obliterated the next time the package is built from scratch. Fortunately, there is a facility to include patches in a package that will be applied to the source after it is unpacked. You will want to create a patch from the modifications that you make to any

package. Fortunately, the packager greatly simplifies the process of making patches.

While making modifications, it's possible to view the diff of modified source files and their unaltered versions. In order to do this, you need to have an unmodified version of the source unpacked in another location for the diffs to be made from. To do this from the Source menu, choose “Prepare Source For Work.”

View the files in “diff” mode by choosing the menu item View Diff from the Source menu. This will display the differences between the working source and the original source in the textbox provided.

Many common file operations are available by right-clicking on files or directories in the directory view.

Installing Patches from External Sources

In addition to making your own patches, often there will be updates to pieces of software in the form of patches. These may be available via an FTP site or other Web site.

Applying patches to software is usually a very simple process. When you download a patch for a piece of software, it should be placed in the directory: <*project dir*>/Packages/local. The name of the patch should then be added to the “Patches” section of the lbc for that package. This can be done by opening the package in the Package Editor, selecting “Patches” in the radio group on the LBC page, entering the filename of the patch in the text area, and clicking on the Save LBC toolbar button.

Here is a quick example for the package “make” in a project located at /home/<*username*>/project/mips:

1. Download “make-3.79.patch1” from the Web.
2. Copy “make-3.79.patch1” to /home/<*username*>/project/mips/Packages/local.
3. Start Package Editor and open the “make” package.
4. In the radio button group on the left side, choose Patches.

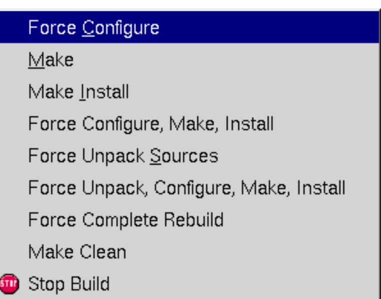
5. Enter “make-3.79.patch1” in the text area. (It is important to enter this patch after any others in the text area and not before because patches are applied in the order that they are listed).
6. Select “Force Complete Rebuild” to make sure that the patch is applied correctly.

Building the Binary Image

Modifying the LBC sections for a package and testing building the package go hand in hand. Whenever an LBC section is modified, you should test building the package to make sure that the changes had the desired affect.

It is possible to do test builds by opening Target Wizard and forcing a rebuild of the package, but this can be cumbersome and does not allow a lot of flexibility (building in Target Wizard is pretty much an all or nothing approach). Package Editor offers a number of building options, all available under the Build menu.

Figure 2-9. Build Options (Build Menu)



Whenever you want to test a phase of building, simply choose that phase from the Build menu and the builder will execute all phases needed up to the specified phase. For example, if this is your first build and you choose “Make,” the builder will extract the source, apply patches, run the configure step and then execute the Make Build instructions. If you choose “Make” again, the builder will only execute the Make Build instructions because the previous steps have already been done.

Here is a list of the available options:

- **Force Configure -** Forces the builder to execute the Make Configure section of the LBC. If you have previously executed the Make Build section, then the builder will do a Make Clean to clean the source before doing the Make Configure.
- **Make -** Executes the Make Build section of the LBC.
- **Make Install -** Executes the Make Install section of the LBC.
- **Force Configure, Make, Install -** Executes a Forced Configure, Make Build, and Make Install all in one.
- **Force Unpack Sources -** Erases the build directory, unpacks the source files from scratch, and applies the patches specified in the LBC.
- **Force Unpack, Configure, Make, Install -** Erases the old build directory and then executes a Forced Configure, Make Build, and Make Install all in one.
- **Force Complete Rebuild -** Erases everything and builds the package from start to finish, just like the builder would inside of Target Wizard.
- **Make Clean -** Executes the Make Clean section of the LBC.
- **Stop Build -** Stops the builder during the build process. It can be used at any time.

Note that in order for any changes made in the LBC to affect a build, the LBC must be saved prior to executing the build. The output of the build appears in the log window at the bottom of the packager. Package Editor uses the Target Wizard Command Line (`twcl`) utility to execute builds so it is using the same build mechanism that Target Wizard uses. Whenever a build is started, the first line that appears in the window will be the actual command used to start the build with `twcl`. Here is an example:

```
+ /usr/bin/twcl --projdir /opt/Embedix/home/
   chrisb/project/intel --pkg hello --force
   --batch --nuke
```

This allows the user to see exactly what arguments are being passed to `twcl`. For more information on using twcl, execute `twcl --help` at the command line to see usage instructions.

In addition to the `twcl` command, all output from the build process will appear in the log window. When a build is finished, an appropriate message will appear at the bottom of the log window. During the build, the user may continue to work with Package Editor without needing to wait for the build to finish.

Sometimes you will need to test a build with ECD options set a specific way. When this situation occurs:

1. Open Target Wizard.
2. Enable or disable the ECD nodes as desired.
3. When finished, right-click on the package component node and then choose Build Component. This will force Target Wizard to write the ECD settings to a file used by the builder.

   At that point you can continue testing the build from Package Editor and the new settings will take affect. Target Wizard does not need to complete it's build as the settings are written out prior to the build. This means that for long packages, you can stop the build in Target Wizard immediately after starting it.

Installing the Package

Package Editor automatically places compiled and configured files in their correct location in the filesystem. (Each package has its own staging area where it waits to be deployed.)

For instructions on installing the binary image (or target image) on a target, refer to the target-specific deployment instructions found in either the *Target Wizard User's Guide* or the board support package (BSP) documentation.

Distributing Packages as LPF Files

Once you have a working package for your project, you may want to distribute it to others or use it in other projects on your system. To simplify this task, Lineo supports a special package format called LPF (Lineo Package Format) files. These files are simply gzipped tarfiles with a control file inside of it. You can use Update Wizard (downloadable from the Lineo Web site or standard as part of the SDK 2.4) to install a package LPF file into a project to have the package appear in the project in Target Wizard. Fortunately, the Package Editor allows you to easily create an LPF file for any package.

To create an LPF from a package:

1. Open the package in Package Editor.
2. From the File menu, choose Make LPF.

   A dialog box appears asking for four different fields:

   - **Version -** This is the version of the package.
   - **Revision -** This is the revision of the package.
   - **Vendor -** This is the entity that this package belongs to. You can enter your name or the company you work for.
   - **Support Contact -** This should be an e-mail address of who to contact if there are problems with the package.
3. When you have filled in those fields, click OK and the LPF file will be created.

   A dialog box appears telling you if the creation was successful and showing the location of the LPF file. Normally it will be placed in the project directory of the project that the package was taken from. This file can then be distributed to other Target Wizard users. They can then install the package into their own projects using Update Wizard.

Using the Embedix Tool Chains

A tool chain is a collection of platform-specific compiler tools. The supported platforms are 8260, ARM, i386, MIPS, PowerPC, and SH.

Note: The best way to leverage a tool chain available in the Embedix SDK is to create an SDK package from existing source code.

This section outlines issues with using the GNU compiler tools included with the Embedix SDK. The tool chains, as set up for Target Wizard, have two modes of operation: Normal and Spoof. Each of these has two variants (explained below).

The Normal mode is used by packages that have awareness of the cross compilation issues. The Spoof mode is used by packages that have no awareness of cross compilation issues.

In order to determine which mode to use, you must answer some questions:

1. *Is your package cross-compiler aware?* In other words, does it know how to distinguish between the use of the host compiler and the cross compiler? This is typically accomplished by defining a couple of values within the Makefile. For example:

   CC=gcc for the host compiler and CROSS_CC=powerpc-linux-gcc for the cross compiler.

   Yes: If the answer is yes, then the Normal mode is the desired mode of operation.

   No: If the answer is no, then you have a couple of options: You can convert the package, making it aware (sometimes this is preferable), or you can attempt to spoof the package and cross compile it (this does not always work and will require that the package be made at least partially aware).

2. *If the Spoof mode of tool chain usage is wanted, does the package build intermediate executables? If yes, do any of the binaries need to be installed on the target?*

If the answer to the first question is yes, then the package must be made partially aware of cross-compile issues (you need to be able to use a native compiler to properly build these executables).

If the answer to the second question is yes, then you need to build two versions of the binaries: the first one for a host-based version and the second one for a target-based version.

3. *Is the package self-contained or does it depend on external libraries (including glibc libraries)?*

   Target Wizard and the tool chains support the ability to create different projects in which different versions of packages are compiled. This means that in package A a minimal subset of glibc can be built and installed. In package B a full blown implementation of glibc can be built.

   When a library package is built, it usually installs:

   public headers into <*project_name*>/build/dev-image/usr/include

   dynamic libraries into <*project_name*>/build/dev-image/lib

   static libraries into <*project_name*>/build/dev-image/usr/lib

   Dynamic libraries also need to be installed on the target machine in the correct location. If you want an application to build using the tool chains, then you need to set three environment variables appropriately—PATH, INCLUDE_PATH_DEFAULT, and LIBRARY_PATH_DEFAULT. These variables control the tool chains' behavior with regard to where executables, SDK header files, and SDK library files are found.

CHAPTER 3 Configuring & Using Metrowerks CodeWarrior with Embedix SDK

This guide describes steps necessary to most effectively use Metrowerks CodeWarrior with Lineo's Embedix SDK toolchain.

Included in this chapter are:

- Embedix SDK installation options
- CodeWarrior installation options
- Embedix SDK configuration
- CodeWarrior base-line configuration
- CodeWarrior per-project configuration.
- Using CodeWarrior with Embedix SDK instructions

Throughout this document, we use the MIPS configuration examples. Other architectures are configured identically, except that the architecture specific tools are selected.

This guide is not intended to be a replacement for the Metrowerks CodeWarrior installation manuals, instead it is designed to augment those manuals by providing Embedix SDK specific information.

Configuring CodeWarrior for Lineo Embedix SDK 2.x

Install and configure CodeWarrior as outlined in the sections that follow.

Embedix SDK Installation Options

Installation of the Embedix SDK for support of CodeWarrior requires no special steps except to ensure that you've selected the "Would you like to add SDK Support for CodeWarrior IDE?" option. This will, among other things, install nine semi-preconfigured CodeWarrior Stationery's. CodeWarrior Stationery is described in

detail later in this document, but it is basically a configuration that allows CodeWarrior to make use of the SDK toolchain. Each Stationery has three possible configurations: one for C++ applications, a second for C applications, and a final configuration for Kernel Modules.

Because configuration of CodeWarrior to use the SDK is project specific, these nine installed Stationery's are not fully ready for development of applications for the SDK. Some of the options will have to be manually configured.

The instructions on configuring CodeWarrior for use with the SDK later in this document detail how to make a configuration from the beginning. If you use any of these nine Stationery files installed by the SDK, many of these options described are already configured. This document will alert you to which options need to be manually configured and which options are preconfigured.

Each of the Stationery's is based on one of nine Lineo cross-compilers. When you open a new project, you must know which cross-compiler is used in your project and select the appropriate Stationery accordingly (how to determine your cross-compiler and how to select Stationery are both described at later points in this document). The nine Stationery's are listed here:

- i386-linux_Remote
- mips-linux_Remote
- mipsel-linux_Remote
- arm-linux_Remote
- sh3-linux_Remote
- sh4-linux_Remote
- m68k-coff_Remote
- m68k-elf_Remote
- powerpc-linux_Remote

We anticipate making additional stationery available from a Web site download as it becomes available.

Post-install SDK Configurations

Some earlier releases of the Embedix SDK will not have a pre-configuration for CodeWarrior execution from the Target Wizard, Run Wizard. Fortunately, the SDK Run Wizard menu is rather easy to configure via a simple python script which when placed in /opt/Embedix/enbedix-2.0/wizards is launched by the Run Wizard.

The script below adds CodeWarrior support to the Run Wizard menu.

Tip: You can use a modification of this script to support other commonly used applications such as editors or DDD.

```
#DESCRIPTION:CodeWarrior 6
#
# The description above appears in the Target Wizard
# "Select Wizard" dialog box.
#
# This wizard shows how to launch a separate application
# from the Target Wizard program.
#
# In order to customize this example, modify:
#       - the global variables 'command' and 'args' .
#       - the DESCRIPTION comment above
# Place the modified script in:
#       /opt/Embedix/embedix-2.0/wizards
# Launch it from File, Run Wizard menu item.
#
# This has several examples of other things that might be of
# interest for programs launched from the SDK, including:
#  - determining the current project directory
#  - adjusting arguments based on arbitrary runtime criteria
#      (in this example, base on whether we could identify the
   proj dir)
#  - setting an environment variable for the launched program

#from wizard import *
import os
import PyProjAPI

command="/usr/local/bin/cwide"
```

```
args=""
sequence=[]

def run_wizard(sequence, globals, standalone=0):
        global command, args

        # get the project directory for the current project
        (r, value) = PyProjAPI.project_get("project dir")
        if r=="OK":
                proj_dir = value
        else:
                proj_dir = None
        # shown for example purposes.  Dont' write stuff to
    stdout normally.
        print "project directory=", proj_dir

        if proj_dir:
                #
                args=proj_dir
                #

        # set an environment variable (just for demonstration)
        os.environ["PATH"]=os.environ["PATH"]+":/usr/local/
    some-other-ddd-dir"

        print 'running command: "%s %s &"' % (command, args)
        os.system("%s %s &" % (command, args))
        return
```

This is the end of the CodeWarrior Run Wizard Python Script.

CodeWarrior Installation and Initial Configuration

This section describes how to install CodeWarrior and then to configure convenience items (e.g. menu items) for best integration with the host Linux desktop and the Embedix tool chain.

Mount your CD-ROM with the mount command (e.g. mount /mnt/cdrom) and then read the README file for instructions on installing this product. Although these installation steps are summarized below, we encourage you to first read the CodeWarrior README file.

1. Log in as root.
2. Mount the CD:

   /mnt/cdrom/install.sh
3. De-select DDD 3.3 (Embedix SDK includes DDD v3.3 located at /opt/Embedix/usr/X11R6/bin/ddd.)
4. Select JDK as required for your specific product development.
5. Select Desktop menu items (KDE/Gnome).
6. Begin complete install.

Note: Pay attention to the warning and do not run CodeWarrior from root.

7. Exit installation procedure.
8. Log out and log back in as a regular user.
9. Configure the new CodeWarrior menu on the KDE start button to point to /usr/local/bin/cwide (instead of cwide6).

Resolving Problems with CodeWarrior Installation

CodeWarrior 6.0 appears to have a bug during installation on some Linux host systems (such as RedHat 7.1). You should first attempt to install according to normal procedure:

```
su
mount -o exec <cdrom directory>
<cdrom directory>/install.sh
```

If this is unsuccessful, follow the procedure below:

```
su
mount -o exec /mnt/cdrom
cd /mnt/cdrom/RPMS
rpm -Uvh --nodeps codewarrior-ide-6.0-b0046d.i386.rpm
```

and optionally:

```
rpm -Uvh --nodeps codewarrior-docs-6.0-b0046d.i386.rpm
rpm -Uvh --nodeps codewarrior-java-6.0-b0046c.i386.rpm
```

```
rpm -Uvh --nodeps codewarrior-jdk11-1.1.8-2.i386.rpm
rpm -Uvh --nodeps codewarrior-jdk12-1.2.2-4.i386.rpm
rpm -Uvh --nodeps codewarrior-jdk13-1.3.0-3.i386.rpm
rpm -Uvh --nodeps codewarrior-mwCVS-1.0-b0636.i386.rpm
```

then

```
/mnt/cdrom/install.sh
```

(this will ensure that the license file has been installed).

Complete the installation as described previously:

1. Exit installation procedure, but remain as root.
2. Enter these commands:

   ```
   cd /usr/local/metrowerks/Lineo/
   ./install_cwide.sh
   ./install_userprefs.sh
   ```

3. Log out and back in as primary user.
4. Configure the new CodeWarrior menu on KDE start button to point to /usr/local/bin/cwide (instead of cwide6)

Common Global Configuration Options for CodeWarrior

These options apply to global settings of CodeWarrior and may make the IDE experience more comfortable for your use.

Although the configuration options are discussed within the CodeWarrior documentation set, we've highlighted the key configuration options here.

External Editor:

If you wish to use a third-party editor (such as kate, kwrite, or Visual Slick Edit (TM))other than the default CodeWarrior editor, you need to complete the following three step process.

1. As root, copy the file

   ```
   /usr/local/metrowerks CodeWarrior_Pro6/CodeWarrior4.1/
   (Helper_Apps)"/External_Editor.lineo
   ```

   to

/usr/local/metrowerks/CodeWarrior_Pro6/CodeWarrior4.1/(Helper_Apps)"/External_Editor

Because this replaces the default file, it is advisable to back up the original External_Editor file before replacing it.

2. Also as root, edit the contents of the newly replaced External_Editor file. It's contents include a list of common editors (all prefixed by “EDITOR=”). The Editor that CodeWarrior will use is the one that is not in comments (i.e. prefaced by a '#'). By default, “kwrite” is uncommented and several other choices are included in comments. Uncomment the editor desired (remove the leading '#') and put kwrite in comments (prepend a '#'). Or, if you wish to use an editor not listed, put kwrite in comments and add a new line “EDITOR=<your editor name here>”.
3. In CodeWarrior (no longer as root), go to: EDIT --> PREFERENCES --> GENERAL --> IDE XTRAS and select the “Use External Editor” option.

Key Bindings:

If you want to change the default key bindings (e.g. cut = alt-x) to more familar bindings (e.g. cut = ctrl-x):

1. Go to: EDIT > COMMANDS & KEY BINDINGS

2. Alter the key bindings as necessary

Internet Browser:

To configure CodeWarrior to select the correct internet browser for you host system (e.g. netscape, mozilla, konqueror, opera, etc.):

1. Go to: EDIT > PREFERENCES > HELP PREFERENCES.
2. Replace the listed browser with a new browser of your choice. The new browser must be accessable on your system path.

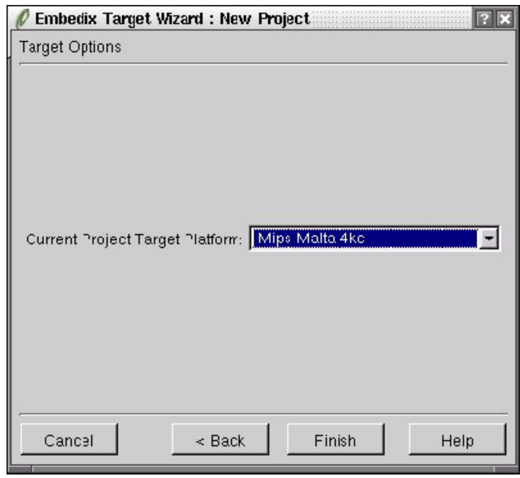
Recording Embedix SDK Tools Setting

Because Lineo's development model is project based, that is, each project is associated with a specific BSP that, itself, has a specific set

of development tools. This step retreives project specific tools information to assist in configuring CodeWarrior.

1. Start Lineo's Target Wizard (enter tw at the command prompt). A screen similar to Figure 3-1 appears.

Figure 3-1

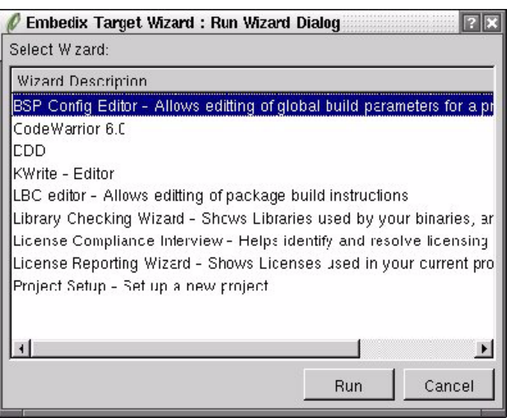


2. Open a new (or existing) project, ensuring to select the proper "Current Project Target Platform" at the Target Options dialog.

Note: The Embedix SDK supports architectures via individual board support packages that are shipped independently from the SDK.

3. Once the new project has been opened, choose the menu item File > Run Wizard. A dialog box like Figure 3-2 appears.

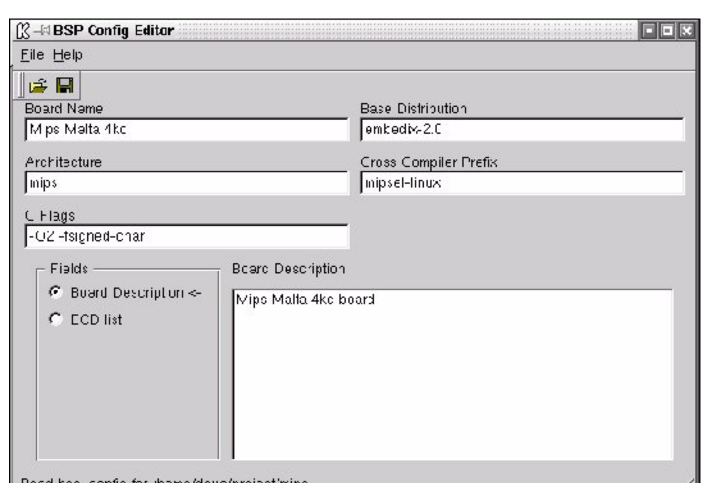
Figure 3-2



4. Run the “BSP Config Editor” and write down the value of the “Cross Compiler Prefix”.

   In the example provided in Figure 3-3, the Cross Compiler Prefix is “mipsel-linux”. As we show additional screens, note that the tools names are common and “as expected” within the Linux development environment except for the addition of this prefix.

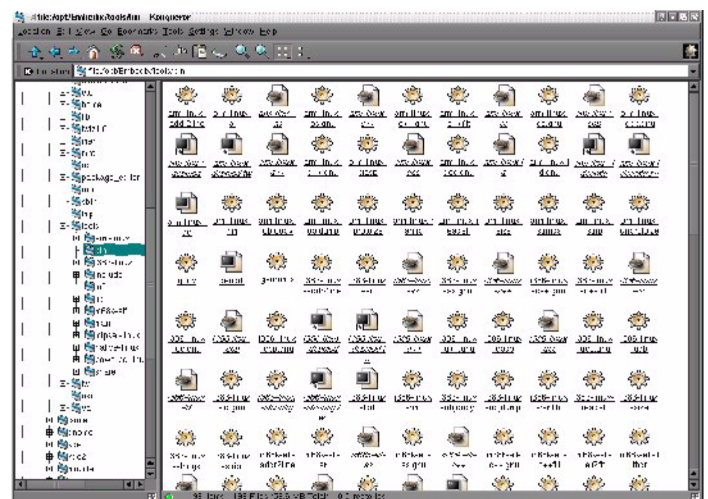
Figure 3-3. Target Wizard BSP Configuration Editor



This cross compiler prefix value is important because Lineo's Embedix SDK places all tools, regardless of architecture, within the same directory (/opt/Embedix/tools/bin) appending a prefix to designate the specific architecture supported by each tool.

Figure 3-4 shows this directory populated with toolchains for multiple board support packages. As you can see by the directory listing below, each of the tools are named identically except for the architecture identifying prefix.

Figure 3-4. Sample /opt/Embedix/tools/bin directory listing



CodeWarrior Per Project Configuration Options

This section describes how to configure CodeWarrior to use the Embedix toolchains for each specific project. A large majority of these items have been pre-configured in the supplied stationery.

Notes:

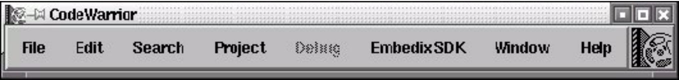
- Documentation/tutorial for CodeWarrior for Linux is located on your system here: /usr/local/metrowerks/CodeWarrior_Pro6/CodeWarrior_Manuals/HTML/Targeting_Linux/UNX000_Front.fm.html
- All of the options described below are set from the CodeWarrior EDIT > TARGET_NAME SETTINGS menu. This menu is located immediately below the "PREFERENCES" menu option and is accessed via the main EDIT menu.
- The GNU tools associated with the Embedix SDK are located in /opt/Embedix/tools/ARCH SPECIFIC directory

Library and Include Files

This section walks through configuration of CW so that the proper libraries and include files are used. These options are set in the SDK installed Stationery's.

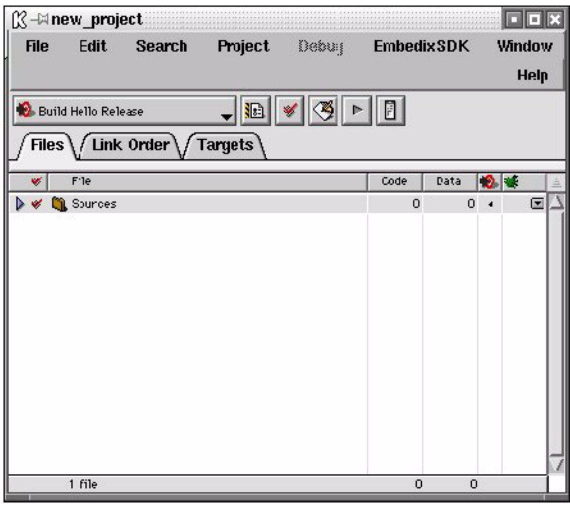
First, start CodeWarrior by entering **`cwide`** at a console, or by using the newly reconfigured desktop icons/menus (see Figure 3-5)...

Figure 3-5. Initial CodeWarrior Start-up Window



... then, open a new project from the File > New menus and their associated dialogs.

Figure 3-6. CodeWarrior New Project Window

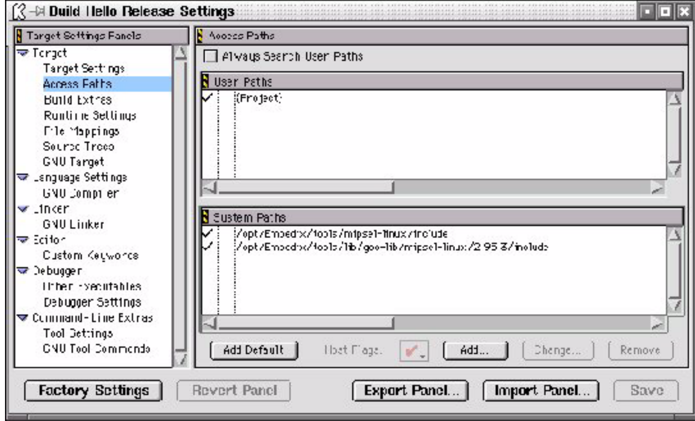


Next, from the <*new_project*> window, select Edit > <*new_project*> Settings > Target > Access Paths.

- Unselect “Always Search User Paths”.

- Select the “System Paths” window > Add > Relative To: “Absolute” > browse and then Add:
- Be sure to add two entries: one for the proper library-include files and the other for the proper header-include files.
- Ensure that the left-most check box on the same row as the files above is checked. (See Figure 3-7.)

Figure 3-7. CodeWarrior Access Path Settings



Note that the base directory for the library include files will generally be: /opt/Embedix/tools/lib/gcc-lib/ CROSS COMPILER PREFIX / RELEASE_SPECIFIC (e.g. 2.95.3)/include.

The base directory for header files will generally be: /opt/Embedix/tools/ CROSS COMPILER PREFIX /include .

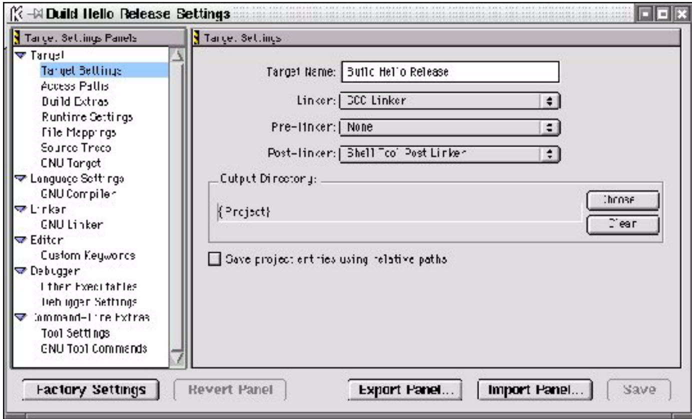
Output Directory

The Output Directory defined by CodeWarrior is where the software is placed following compilation. This option is not set in the SDK installed Stationery’s and must be manually configured.

1. From the <new_project> window, select Edit > Build Hello Release Settings > Target > Target Settings.
2. Select the output directory and file name of choice.

Note: Setting the output directory as any other directory will require you to transfer via NFS, ftp, ssh, etc.

Figure 3-8. CodeWarrior Target Settings



Useful Output Directory Locations:

The CodeWarrior {PROJECT} directory is the default build location.

Configuring the output directory to the Target Wizard project / merge directory (e.g. /home/lineo/project/PROJECT_NAME/ merge) will automatically deploy the application with the rest of the system via the deployment wizard. It will also enable the GPL compliance tool to validate library linking between your new application and the deployable system.

Some BSPs create a /tftpboot directory, which is required for this boot method.

Debugger

This step configures CodeWarrior so that is uses the proper debugger.

At this point, if you choose to use DDD 3.3 which was provided with CodeWarrior, you need to enter nothing as shown in figure 9 below. However, if you encounter problems with DDD 3.3, you may want to use DDD 3.3 provided with the Embedix SDK. Also, if you are using one of the SDK installed Stationery's, they are preconfigured to use the SDK DDD 3.3; if you wish to use the CodeWarrior DDD 3.3, you must turn off using a third party debugger.

To do this:

1. From the <NEW PROJECT> window select EDIT > BUILD HELLO RELEASE SETTINGS > TARGET > BUILD EXTRAS
2. Unselect "Use third party debugger"

To change the configuration to use the SDK DDD 3.3:

1. From the <NEW PROJECT> window select EDIT > BUILD HELLO RELEASE SETTINGS > TARGET > BUILD EXTRAS
2. Select "Use third party debugger"
3. Enter the location and command for the debugger followed by any arguments: /opt/Embedix/usr/X11R6/bin/ddd in the file field.
4. Consult the DDD documentation for a description of all arguments.

   An example for using the CodeWarrior DDD 3.3 with the 5272C3 is provided below:

   ```
   ddd --debugger "m68k-bdm-elf-gdb -n -x /
       tftboot/tarifa/gdbinit" %1.gdb
   ```

   Descriptions:

   --debugger "<debuggercmdline>" tells ddd which gdb frontend to use.

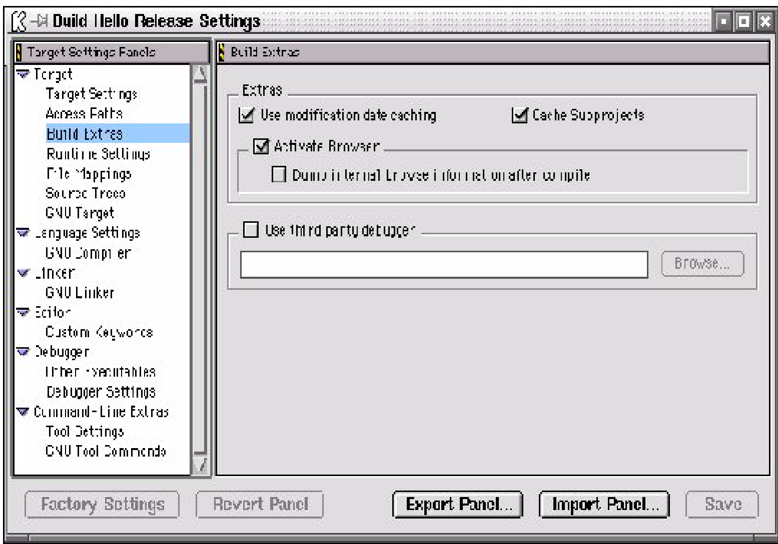
   -n tells the gdb not to invoke any .gdbinit that might be in the execution directory

   -x <file> tells an explicit gdbinit file to use

   %1.gdb is special to m68k-elf as uClinux usualy produces (non-elf)flat binaries plus (non-executable) elf binaries

(with .gdb postfix) so for non uClinux architectures, you may just enter %1 instead

Figure 3-9. CodeWarrior Build Extras Settings



Compiler Command Line Arguments

CodeWarrior is extremely versatile, but it does not eliminate the developer's need to know essential compiler arguments. However, it does greatly simplify the creation of applications and associated project management once these arguments are in place. This section describes how and where to set these command line arguments. Some basic command line arguments are already in place in the SDK installed Stationery's; however, the configurations for Kernel Modules in each of these Stationery's are incomplete out of necessity and even the configurations for C++ and C applications should be reviewed to ensure that the arguments are correct for the application being developed.

Note: Default SDK Global SDK GCC arguments for a specific project can be found in Target Wizard: FILE > RUN WIZARD > BSP CONFIGURATION EDITOR > C FLAGS

Note: Individual package GCC arguments can be found in Target Wizard: FILE > RUN WIZARD > LBC EDITOR > COMPILE FLAGS

Note: In CodeWarrior remember to add in the -o command line argument when compiling kernel modules.

Note: CodeWarrior defaults to -g as the debug command line option.

To set these GCC command line arguments:

1. From the <*new_project*> window, select Edit > Build Hello Release Settings > Language Settings > GNU Compiler.
2. Enter in the GCC Command Line Arguments associated with your specific board. Consult the GCC documentation for a full explanation of these options. Figure 3-10 shows the basic command line arguments for creating a user application.

   Below, we've provided example GCC command and arguments for a kernel module:

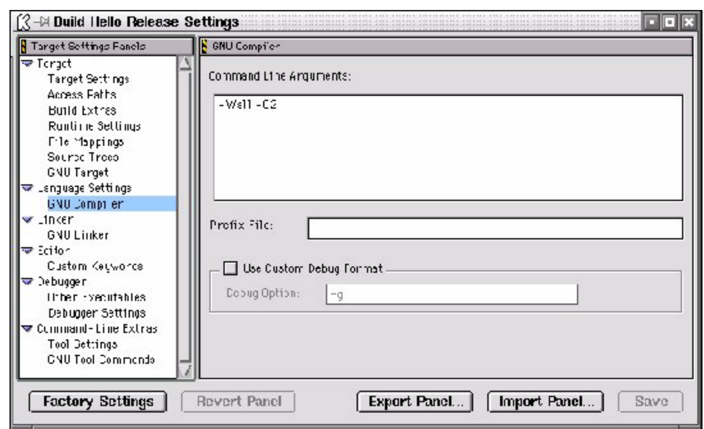
   ```
   -g -D__KERNEL__ -DMODULE -O2 -Wall -
      I<Include file paths>/ -c filename.c -o
      filename.o
   ```

   Descriptions:

   - The **`-g`** option adds in the debug symbols for use with GDB.

- The **-D__KERNEL__** flag (note the double underscores!)tells the preprocessor to select certain parts of kernel headers.
- You must define the **-DMODULE** symbol for a kernel loadable module,and you should define it before including the linux/module.h file.
- You must enable optimization with the **-O2** flag because many functions are declared as inline in the header files,and gcc does not expand inlines unless optimization is enabled.
- We recommend turning on compiler warnings with the **-Wall** flag because eliminating all compiler warnings helps to prevent unexpected errors later on.
- The **-I<Include file paths>** flag specifies the directories in which the included header files can be found.
- The **-c** filename.c option tells gcc to stop after generating the object file (it does not go on to the link phase).
- The **-o** filename.o option tells the compiler to create an object file of name filename .

Figure 3-10. CodeWarrior GNU Compiler Settings



Linker

This section describes how and where to set these command line arguments for the Linker. The SDK installed Stationery's have some linker options preconfigured for C++ applications. However, due to the nature of linker arguments, the preconfigured arguments should be reviewed.

Because gcc performs compilation as well as linking, it is often not required to enter any linker specific arguments as shown in Figure 3-11. To change these settings:

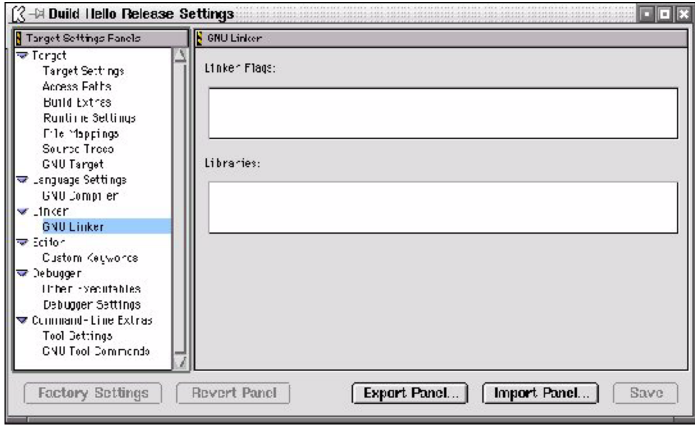
1. From the <*new_project*> window select Edit > Build Hello Release Settings > Linker > GNU Linker.
2. In the dialog box provided, enter the Linker flags associated with your specific board.
3. In the dialog box provided, enter the Libraries.

   Note: Linux for MMU-less devices (uClinux) requires some additional configuration options for the linker. Following is an example of the linker command line arguments for the Motorola ColdFire 5272C3, which form a good starting point for MMU-less configuration:

```
-m5200 -Wl,-elf2flt,"-T /opt/Embedix/
tools/m68k-elf/lib/elf2flt.ld"
```

In the "Libraries" window, enter: `-lc`

Figure 3-11. CodeWarrior GNU Linker Settings



Configure CodeWarrior to Use the Right GNU Tools

This step ensures that CodeWarrior access the proper target specific GNU tools provided with the Embedix tool-chain. This information is set in the SDK installed Stationery's.

1. From the <*new_project*> window, select Edit > Build Hello Release Settings > Command-Line Extras > GNU Tools Commands. A screen similar to Figure 3-12 on page 81 appears.
2. Select “Use Custom Tool Commands” and then complete the dialog boxes provided on the right of the window:

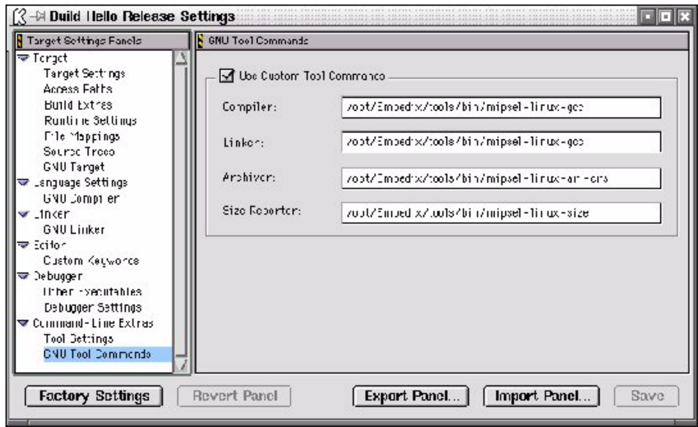
   Compiler: Point to /opt/Embedix/tools/bin/CROSS COMPILER PREFIX-gcc

   Linker: Point to /opt/Embedix/tools/bin/CROSS COMPILER PREFIX-gcc

   Archiver: Point to /opt/Embedix/tools/bin/CROSS COMPILER PREFIX-ar crs

Size Reporter: Point to /opt/Embedix/tools/bin/CROSS COMPILER PREFIX-size

Figure 3-12. CodeWarrior GNU Tool Commands Settings



Configure CodeWarrior to Recognize *.ecd and *.lbc Files

CodeWarrior will not recognize files which have an unknown or undefined (to CodeWarrior) extension. If you want to edit Lineo's ECD and LBC file formats via CodeWarrior, you need to configure the File Mappings Page so that these are recognized. This information is set in the SDK installed Stationery's. (See Figure 3-13.)

Figure 3-13. CodeWarrior File Mappings Configuration

1. From the <new_project> window select Edit > Build Hello Release Settings > Commannd-Line Extras > File Mappings.
2. Map .lbc files by completing these steps:
   - **2a.** In the “File Type” box, select “Text”
   - **2b.** In the “Extension” box, enter “.lbc”
   - **2c.** In the Compiler entry, select “None”
   - **2d.** Click “ADD”
3. Repeat the process for .ecd files:
   - **3a.** In the “File Type” box, select “Text”
   - **3b.** In the “Extension” box, enter “.ecd”
   - **3c.** In the Compiler entry, select “None”
   - **3d.** Click “ADD”
4. Repeat the process for .in files (gdbinit.in):
   - **4a.** In the “File Type” box, select “Text”
   - **4b.** In the “Extension” box, enter “.in”
   - **4c.** In the Compiler entry, select “None”
   - **4d.** Click “ADD”

Creating Project “Stationery” Files

CodeWarrior includes project Stationery which are templates which store all of the information that you’ve just set up. These files allow you to avoid this set-up procedure for each new project. If you are using the SDK installed Stationery's, this may or may not be necessary depending on the nature of the projects involved. Most of the configuration not in the SDK Stationery's are project specific and may not transfer well to other projects. This section also contains instructions on accessing and selecting Stationery.

To select a stationery file:

From the initial CodeWarrior start-up window (Figure 3-5), you can select a stationery by: File > New > (choose any stationery).

To create a new stationery file:

1. As root, ease the write priviledges on /usr/local/metrowerks/CodeWarrior_Pro6/CodeWarrior4.1/(Project_Stationery) to allow the user to write files.

   ```
   chmod -R 777 * "/usr/local/metrowerks/CodeWarrior_Pro6/
       CodeWarrior4.1/(Project_Stationery)"
   ```

   (It should be noted that quotes or '\'s may be necessary to access the “(Project_Stationery)” directory because of the parenthesis).

2. Create a new directory named “<ARCH>_Remote” under: /usr/local/metrowerks/CodeWarrior_Pro6/CodeWarrior4.1/(Project_Stationery).

   This is the name which will appear as available stationery when a new project is opened.

3. Underneath the new directory above, create separate directories <TEMPLATE_DESCRIPTION> whose names describe the type of templates you are creating. (e.g. /C_Application, /C_Shared_Library, C++_Application, C_Kernel_Module, etc.) These are the names which will appear as available project types within the architecture selected in Step 2.

4. Start CodeWarrior.

5. Create a new project, selecting any suitable stationery as the starting point.
6. Configure all of the target specific settings as described in this document. Note that the configuration for the Build XYZ Release and Build XYZ Debug are independent. Thus, you need to configure and save them independently.
7. Build the project (remember to build both Release and Debug configurations).
8. Using a name of PROJECT_NAME.mcp, save the project to the appropriate directory: /usr/local/metrowerks/CodeWarrior_Pro6/CodeWarrior4.1/(Project_Stationery)/<ARCH>_Remote/<TEMPLATE DESCRIPTION>

Hints and Notes:

Each of the settings panels includes the EXPORT PANEL option. This allows you to save the settings on any specific panel, to be restored with IMPORT PANEL.

If you can't export the panel, review your settings. Some pages, like the "Target Settings" page, will not allow you to export a panel if something within the panel (such as file name) conflicts with an existing file name or does not exist.

Adding New Tools to the CodeWarrior Menus

Once CodeWarrior is configured for the essential elements, it is often useful to add your own helper applications, or new Lineo applications as they become available to the CodeWarrior menus. This section describes this simple procedure.

1. From the Edit > Command and Key Bindings menu, select the EmbedixSDK command group.
2. Click NEW COMMAND.
3. Enter in the name of the tool (Kwrite, Package Editor, etc.)
4. Enter in command line arguments, etc.
5. Ensure that "Appears in Menus" is selected.

6. Select SAVE
7. Close the window (X).

Using CodeWarrior with Embedix SDK

Lineo has configured CodeWarrior for the majority of architecture specific elements including: cross-compilers, compiler command line arguments, libraries, headers, include files, and debug agents/stubs.

The purpose of this section is to describe the top level details associated with using the CodeWarrior IDE to create applications which can be run and debugged on a target system which was created and deployed with Embedix Target Wizard. It is not meant as a replacement for the full CodeWarrior documentation set which, provided that it was installed, can be found on the host development system at: /usr/local/metrowerks/CodeWarrior_Pro6/CodeWarrior_Manuals/

1. Start CodeWarrior.

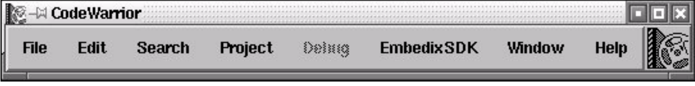
   To start CodeWarrior, simply select the KDE or Gnome menu option which you may have opted to provide during the CodeWarrior installation process.

   Alternatively, you can open an xconsole and type:

   ```
   cwide
   ```

   When CodeWarrior initializes, you will be presented with a window as shown below:

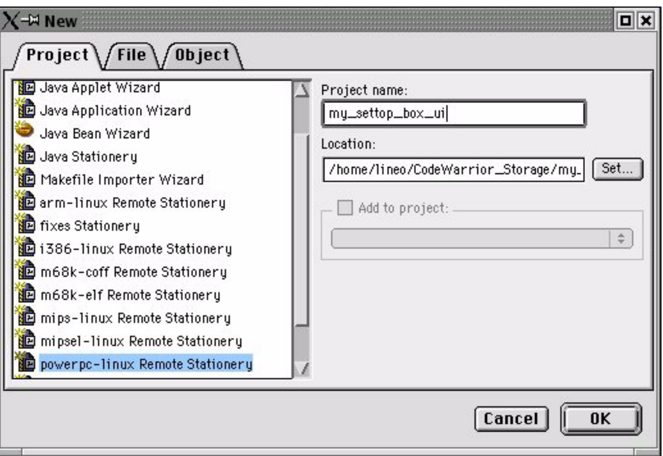
Figure 3-14. CodeWarrior Menu Bar



2. Choose an architecture-specific tool chain.

This step requires you to choose the architecture which coincides with your target hardware and the Embedix board support package.

Figure 3-15. Architecture Options



2a. Highlighted the appropriate architecture.

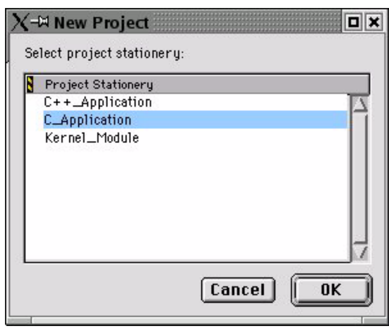
If you are unsure which tool chain is appropriate, start Target Wizard and open your board-specific project. Then, from the menu bar, choose File > Run Wizard and select “BSP Config Editor” and then note the entry listed in the "Cross Compiler Prefix" entry. This value coincides with the options provided in the CodeWarrior dialog shown below.

2b. Enter a project name and location (directory path) and then click OK.

3. Choose the Application Type.

Tell CodeWarrior what type of application you wish to develop as shown below and click OK.

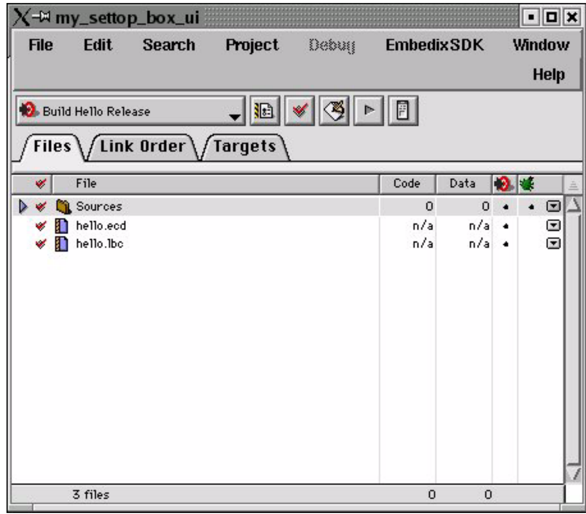
Figure 3-16. Application Type



4. Select a “Target.”

   CodeWarrior uses the term "target" to define unique configurations of compile and debug tools which get applied against the same source files. Lineo has pre-configured three separate targets: Build Hello Self Host- x86, Build Hello Debug, and Build Hello Release, which are selectable from the drop down box located just below the menu bar (see Figure 3-17).

Figure 3-17. Opened Project with “Build Hello Release” Target Selected



- **Self Hosted - x86 Target**

  One of the advantages of developing on a Linux machine is that the API running on the development platform is the same API which runs on the target. Thus, you can develop and debug the large majority of your user space applications on the desktop, even before the development hardware is available. In consideration of this, Embedix includes a Self Hosted -x86 target which builds and debugs against tools for x86 platforms.

  Build Hello Release settings are defaulted to apply code optimization to the compiled application (i.e. gcc -O2 ...). Applications to be debugged should generally not be code optimized.

- **Debug Target**

  Build Hello Debug settings are defaulted to apply no code optimization to the compiled application (i.e. gcc -O0 ...). Applications to be debugged should generally not be code optimized.

- **Release Target**

  Build Hello Release settings are defaulted to apply code optimization to the compiled application (i.e. gcc -O2 ...).

  Note: Even when you have selected Build Hello Release (which has been optimized for non-debug purposes), the compiler will default to including debug symbols as long as the small black circle below the green “bug” is selected. (The green “bug” is seen directly above the main pane and immediately to the right of the "bullseye" icon.) So, be sure to de-select this icon.

5. Set the Output Directory and Target Name.

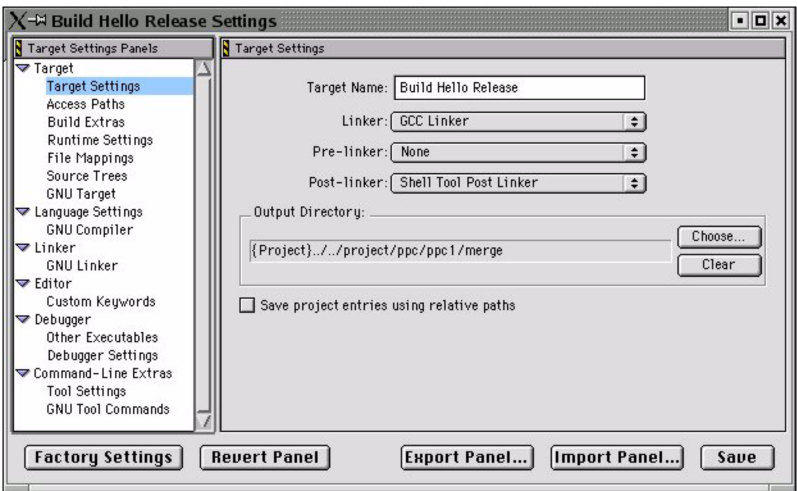
   The output directory, that is, the directory where the compiled application (or kernel module) is placed is project dependent. This directory can be set to one of several locations depending on how you plan on developing and deploying your software.

Typical deployment options include:

- **Project specific /merge directory:** The recursive contents of the project specific /merge located at: /home/USER NAME/ project/PROJECT DIRECTORY/PROJECT.
- NAME/merge are included in the identical locations within the target's root file system upon deployment. When the user's application is located here, it will receive full benefit of the Embedix SDK tools, including Lipo, GPL Tool, and automatic deployment onto the target.
- **tftpboot directory:** When the tftpboot deploy method is used, software is initially located on the host under a directory traditionally named /tftpboot. This software is then moved over to the target once the system has booted.
- **NFS mount directory:** NFS mounts are physically located on the host development platform, but mounted via the network and treated as if they were physically located on the target.
- **Other defined directory:** In cases where the Self Hosted -x86 target it used, the developer may choose any arbitrary directory location. Other directories may be appropriate for alternative deployment schemes.

Unless you plan on deploying the hello world application, you'll probably want to rename your target. This is performed from the same dialog as used to set the output directory (see Figure 3-18).

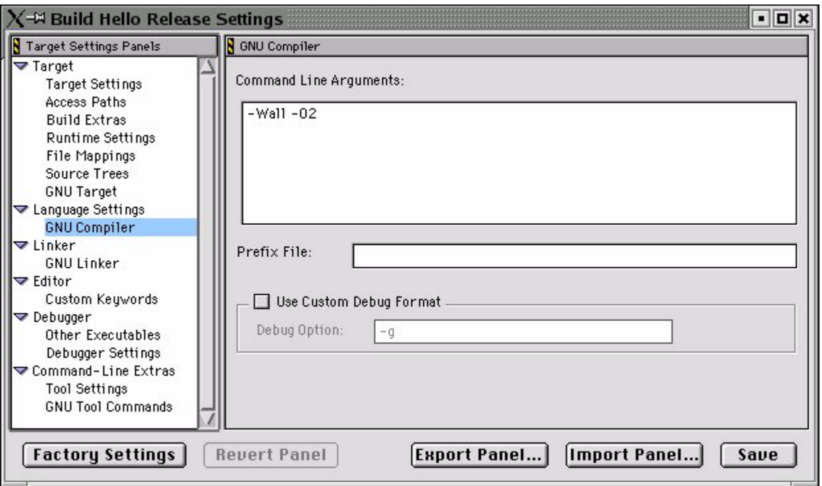
Figure 3-18. Output Directory



6. Review the GCC Arguments

   We've pre-configured the compile and debug options for the majority of uses, but it is probably worthwhile for you to review them and custmize as needed.

Figure 3-19. GCC Arguments



7. Edit your Source Code.

   To Edit your source code, simply expand the “Sources” tree and double-click the file which you would like to edit. If you've performed the CodeWarrior configuration steps as described in this document, your application will open up in your favorite editor.

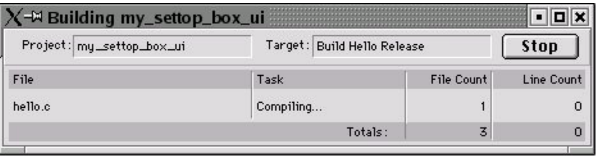
Figure 3-20. Sample View of Editor

```
/*
 *  Hello World for the CodeWarrior
 *  © 1997-1998 Metrowerks Corp.
 *
 *  Questions and comments to:
 *       <mailto:support@metrowerks.com>
 *       <http://www.metrowerks.com/>
 */

#include <stdio.h>

int
main(void)
{

    printf("Hello World, this is CodeWarrior!\n\n");

    return 0;

}
```

8. Make/Build

   CodeWarrior dramatically eases the make/build process. Simply click the third icon from the target selection (which looks like an envelope with a pencil writing on it). When you do, another window will open showing the status of the compile.

Figure 3-21. Make File Compile Status



9. Test your Application on the Development Platform.

This step shows you how to run and debug your application on the host platform.

9a. Ensure that Self Hosted-x86 Target is selected.

9b. Make/Build the application.

9c. From the menu bar, choose Project > Debug. This will start the DDD (debugger) interface and connect it to the architecture specific gdb.

Figure 3-22. DDD (Debugger) Interface

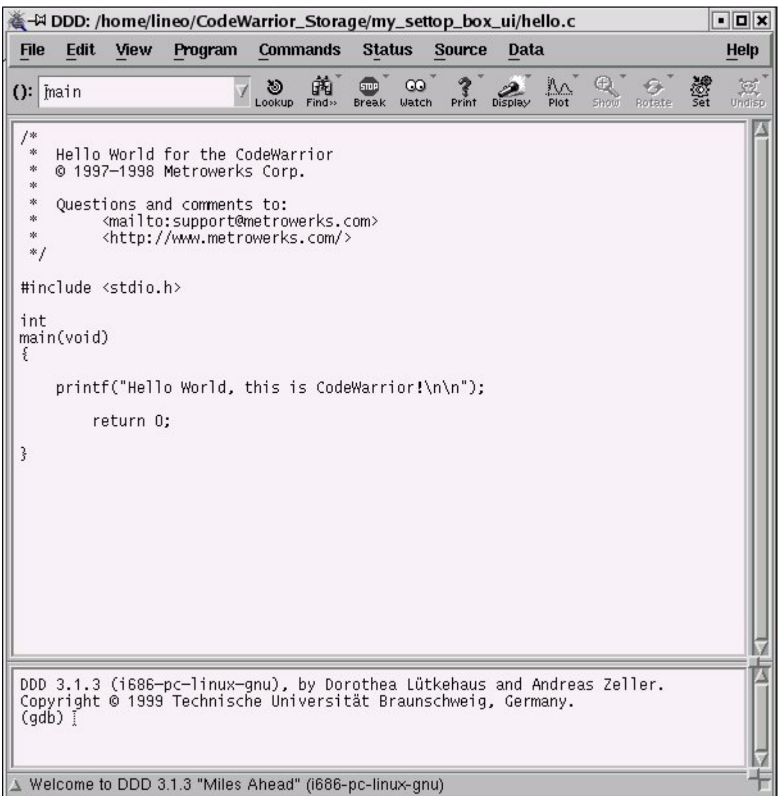


Figure 3-23. DDD (Debugger) Menu

10. Change “Targets” for the Development Hardware.

Now that your application runs on the x86 platform, select the Debug Target and make/build the application.

11. Test your Application on the Target Platform.

This step shows you how to run and debug your application on the development hardware.

11a. Ensure that your target has been configured with gdbserver via Target Wizard.

11b. Open a console (remotely or natively) on the target platform and start gdbserver.

11c. From the menu bar, choose Project > Debug. This will start the DDD (debugger) interface and connect it to the architecture specific gdb.

11d. It may be convenient for you to use a gdbinit file which is located in the same directory output directory, or in a separate directory. To configure this, navigate to the CodeWarrior Target settings Build Extras > Third Party Debugger dialog, then enter:

```
/opt/Embedix/usr/X11R6/bin/ddd --debugger "opt/
Embedix/tools/bin/ARCH-linux-gdb -x -n GDBINIT_DIR/
gdbinit" %1
```

See the DDD and GDB manuals for detailed information on how to debug using these tools.

CHAPTER 4 Debugging Using GDB and DDD

This chapter describes using the GDB and DDD debugging tools on your target image after the target image has been deploy to your target.

The GNU Debugger, GDB, is a debugging tool that provides source-level run-time debugging. It is used during development to aid in finding and fixing problems.

The Data Display Debugger, DDD, is a graphical user interface for GDB. Effective debugging is easy using the graphical interface provided by DDD rather than by using the command-line driven GDB by itself. The DDD interface provides menus for GDB functions and windows for GDB commands, code listing, and variables. It also automatically runs GDB commands in the background to provide useful information.

Note: When the cursor is positioned over an item of interest in the source code, the variable types and values of that item are displayed.

Understanding Remote Debugging

The word *Host* refers to the development host machine or system that is used to develop the system software that will be deployed to the target. This is the system that you run Embedix Target Wizard on to create the target image.

The word *Target* refers to the target device or system to which you will deploy your newly developed software.

The supported debugging method is referred to as host-target debugging, because GDB is used on the host system communicating over a communication port to either gdbserver or gdbstubs running on the target system.

General Steps to Debugging a Target Kernel or Target Application

1. Prepare your target image using Embedix Target Wizard, configuring it as needed for your debugging needs.

   For example, you may to do one of the following from the tree view of your target project:

   - Kernel debugging requires startkgdb to be enabled.
   - Kernel module debugging requires kernel module support to be enabled.
   - Serial ports or network drivers needed for communication need to be enabled in the target kernel. Application debugging can become more efficient when IP network communications is enabled.
2. Deploy an OS that has been configured for debugging to your target.
3. Connect a serial and/or networking cable between target and host machines.
4. Configure host and target software for debugging communication, including exporting appropriate paths.
5. Debug.

The following tutorials are included to help you set up a variety of embedded debugging tasks that may be new to some software engineers.

"Overview of Debugging an Embedix Target Image" on page 101
"Kernel Debugging at Boot Time" on page 102
"Kernel Debugging—Started from the Shell" on page 105
"Kernel Module Debugging" on page 106

"Application Debugging over IP" on page 109
"Application Debugging over Serial Port" on page 111

Using DDD and GDB from Target Wizard's Tools Menu

Launching DDD

The SDK 2.4 features the new 'Tools' menu in Target Wizard. From this menu you are able to launch DDD and have the SDK automatically attach it to a GDB for your board.

The SDK 2.4 installs DDD version 3.3.1 under /opt/Embedix/usr/X11R6/bin/

When you launch DDD from Target Wizard, it will use the DDD installed with the SDK. After DDD is launched, it will attach to a specific GDB based on your current project settings (DDD cannot be launched unless a project is open).

GDBs Used by the SDK

Under most circumstances, GDB needs only be configured against the cross compiler used for a target board. The SDK has nine preconfigured GDB's located under /opt/Embedix/tools/bin/. These nine GDB's are listed below:

arm-linux-gdb
m68k-elf-gdb
m68k-coff-gdb
powerpc-linux-gdb
sh3-linux-gdb
sh4-linux-gdb
i386-linux-gdb
mips-linux-gdb
mipsel-linux-gdb

Each GDB includes the name of the cross-compiler it is configured for in the name. These are the most common nine cross-compilers used in the SDK.

However, if your project uses a cross-compiler not listed above, or if the default GDB does not work for your target, you have the option of building your own GDB binary and placing it under <project-directory>/emb-bin/ (it must be named 'gdb'). Some Lineo BSP's have special GDB binaries which are placed in this directory for each project. Please refer to each BSP's documentation for details.

In summary, DDD will first look to connect to a GDB under <project-directory>/emb-bin/ (it will expect the file to be named 'gdb'). If it cannot find GDB there, it will revert to using the cross-compiler specific GDB found uner the /opt/Embedix/tools/bin/ directory.

Manual Execution of DDD

The Target Wizard option to launch DDD is available for convenience in working with the most common debugging tasks. However, you may wish to have more direct control over starting DDD (e.g., you may wish to specify it's command-line arguments, etc). Many BSP documentations have instructions for using ddd and/or gdb from the command-line.

If DDD is already installed on your system, execute the debugger by entering 'ddd' at a command-line prompt. If you do not have DDD on your system, or wish to use the DDD provided with the SDK, enter '/opt/Embedix/usr/X11R6/bin/ddd' at the comand-line. In either case, command-line arguments may be used as desired (the '--help' option lists all command-line arguments).

Using DDD

DDD has detailed documentation about it's operations and feature-set. Included on the SDK 2.4 CD is a DDD manual, GDB manual and a DDD tutorial. The locations and filenames are listed below:

- <*cdrom directory*>/EmbedixSDKdddManual/pdf/ddd-3.3.1.pdf
- <*cdrom directory*>/EmbedixSDKdddManual/pdf/gdb.pdf

- <*cdrom directory*>/EmbedixSDKdddManual/pdf/ddd_debug_tutorial.pdf

For convenience, the following instructions are included for remote debugging of an application.

1. Deploy the target application

   Before you can debug your application on the target hardware, you must first deploy the application to the target. Please refer to the Target Wizard manual and your BSP documentation. You must also deploy gdbserver (PROGRAMMING > DEBUGGING > GDBSERVER).

   If you wish to debug over TCP/IP instead of serial, you must make sure that the package 'iproute2' is enabled in your deployed filesystem. Package 'iproute2' (SYSTEM > IPROUTE2). You must make sure that this package is built and deployed or you will not be able to debug over TCP/IP.

2. Boot the target and start gdbserver Boot the target hardware. From a console window, launch gdbserver:

   ```
   gdbserver localhost:<port> <full-path-to-application>
   ```

3. On the host, launch ddd.

   To launch ddd, see the previous instructions on launching ddd from the Tools menu in Target Wizard.

   Once ddd is started, load the program that you wish to debug. This means that you must have the application on both the target and the host. Theapplication can be stripped on the target, but must NOT be stripped on the host. If you wish do debug from source code, you must also have the source code on the host and in the same directory as the binary.

   If you have built your application using Target Wizard, you will find your source and binary under your <project_directory>/src/<application_name> directory.

   To load the program in ddd, select FILE > OPEN PROGRAM menu. Browse to the appropriate directory, and select the _binary_ file. ddd will load the appropriate source code.

4. Connect to the remote gdbserver

   At the bottom of the ddd GUI is the gdb console. At the '(gdb)' prompt, type 'Target Remote <gdbserver's IP address>:<port>'.

5. Debug the application normally.

   Once you are connected to the remote gdbserver, you can debug the application as you would debug it locally (e.g. breakpoints, watch's, etc).

   In addition to the remote configuration, you may wish to configure some global options of your ddd. The following descriptions are found in the ddd documentation, but are included here for convenience.

Editor Configuration

1. In DDD, browse to the following menu:

   Edit > Preferences > Helpers > Edit Sources

2. Enter "<*EDITOR_NAME*> @FILE@" (where <*EDITOR_NAME*< is replaced with the actual name of the editor of choice e.g. "kate @FILE@"). If the editor desired is not on the path, the absolute path to its executable must be entered.

   Note: Some editors also have the ability to jump directly to the line of code selected in DDD. If your selected editor is capable of this, enter "<*EDITOR_NAME*> @FILE@ @LINE@"

3. Press the HELP button from the HELPERS screen for more information.

4. When finished, select Edit > Save Options.

Web Browser Configuration

1. In DDD, browse to the following menu:

   Edit > Preferences > Helpers > Web Browser

2. Enter the name of the web browser desired. If the web browser desired is not on the path, the absolute path to it's executable must be entered.
3. Then select Edit > Save Options.

Tip of the Day

To turn off(/on) tip of the day:

1. In DDD, browse to the following menu: Edit > Preferences > Startup.
2. Unselect(/select) tip of the day.
3. Then select: Edit > Save Options.

Warning

Any preferences you select under Edit > Preferences will not be kept on the next instantiation of ddd unless you select Edit > Save Options. You must do this or your changes will be lost the next time you run the debugger.

Overview of Debugging an Embedix Target Image

The following is a brief tutorial on debugging an Embedix target image using GDB with DDD and GDB's associated agent programs, gdbserver and gdbstubs. The intent is to provide immediate access to the features of GDB that allow host-target debugging.

Debugging the Target from the Host

Usually GDB is used by itself on a single computer. In host-target debugging, GDB is used on the host machine that was used to develop the software image that was moved to the target system. To provide this ability, GDB communicates with an agent program running on the target system. A few simple GDB commands make this possible.

Embedix SDK provides a copy of GDB for each processor architecture in your chosen support package. These executables are

located at
/opt/Embedix/tools/bin/. (These versions of GDB will not conflict with the copy of GDB provided in your host system.)

For GDB to debug a program running on your target board, it needs a debug agent running on the target while GDB runs on the host system.

Note: When GDB runs on the host, it must have access to the source code and the unstripped application. (Usually a stripped executable is deployed to the target to save space in the target's filesystem.)

Gdb agent programs used on the target include *gdbserver* and *kgdb*.

gdbserver is the user space application debug agent.

kgdb is the kernel debug agent derived from the gdbstub source code. It is sometimes referred to as stub, gdbstub, or kgdb (for *kernel* gdb).

When debugging the target system, run the version of GDB that matches the target system.

This list of options is not comprehensive. The options available are based upon the BSPs you have installed with Embedix SDK. They are located in /opt/Embedix/tools/bin/ on your host system.

The following tutorials assume that the target is an intel i386 processor; therefore, the version of GDB used is "i386-linux-gdb." You can avoid typing the full path for GDB by adding "/opt/Embedix/tools/bin/" to your path environment variable.

Kernel Debugging at Boot Time

Complete the following steps to start a debugging session for your target's kernel:

1. In the Target Wizard tree view,

1a. Enable **Embedix > Kernel > Kernel hacking > Remote (serial)** kernel debugging with GDB.

1b. Enable appropriate serial drivers in the kernel.

Note: By default a stripped kernel has been placed in your target image and an unstripped kernel has been left in the project kernel source directory for use with GDB.

2. Connect a serial cable from the host system to the target system.
3. On the host system, open an X-term and then complete the following steps at the prompt:

3a. Provide paths to the executables by entering this:

```
export PATH=/opt/Embedix/tools/bin/:/opt/Embedix/
usr/X11R6/bin/:$PATH
```

3b. Go to the kernel source directory.

This is under the project directory you created with Embedix Target Wizard. The command will be similar to this:

```
cd /home/<username>/project/<projectname>
/src/linux/
```

Note: In order for GDB to use the symbol information, the program's source code must remain in the same directory path it was in during compilation.

3c. Start the GDB executable for your target processor.

For example, to start GDB, enter this command:

```
i386-linux-gdb vmlinux
```

Alternatively, start DDD with GDB:

```
ddd --debugger i386-linux-gdb vmlinux
```

4. At the gdb prompt (in the GDB command window within DDD) configure the remote connection by entering these two commands:

```
set remotebaud 9600
target remote /dev/ttyS0
```

Note: Some agents have communication routines that time out after lack of communication for a period of time. Because of this, it is best to put GDB into "target remote" mode within about 30 seconds of the time the agent is started.

5. To debug kernel code closer to the entry point, start the stub from the kernel command line.

   Enter the following at the LILO Boot prompt (assuming you have a label of "embedix" within the /etc/lilo.conf file, a serial port connection to COM1, such as ttyS0, and a communication speed of 9600 baud):

```
embedix gdb gdbttyS=0 gdbbaud=9600
```

   You can also do this by adding an append line within the /etc/lilo.conf file.

```
append="gdb gdbttyS=0 gdbbaud=9600"
```

   After running LILO with this line in lilo.conf, the system will start up in debug mode until the line is removed and LILO has been run again.

   The 'permanent' nature of a change like this is a good reason to just use the command line unless you need to reboot many times while debugging the kernel.

   You can now debug the target kernel from the host system. For more information about debuggers and how to use them, see "Additional Resources" on page 113.

Kernel Debugging—Started from the Shell

Complete the following steps to start a debugging session for the kernel:

1. In the Target Wizard tree view,

   1a. Enable **Embedix > Programming > Debugger > startkgdb**.

   1b. Enable **Embedix > Kernel > Kernel hacking > Remote (serial) kernel debugging with GDB**.

   1c. Enable appropriate serial drivers in the kernel.

Note: By default a stripped kernel has been placed in your target image and an unstripped kernel has been left in the project kernel source directory for use with GDB.

2. Connect a serial cable from the host system to the target system.
3. On the host system, open an X-term and then complete the following steps at the prompt:

   3a. Provide paths to the executables by entering this command:

```
export PATH=/opt/Embedix/tools/bin/:/opt/Embedix/usr/
    X11R6/bin/:$PATH
```

   3b. Go to the kernel source directory.

   This is under the project directory you created with Embedix Target Wizard. The command will be similar to this:

```
cd /home/<username>/project/<projectname>
    /src/linux/
```

Note: In order for GDB to use the symbol information, the program's source code must remain in the same directory path it was in during compilation.

   3c. Start the GDB executable for your target processor.

For example, to start GDB, enter this command:

```
i386-linux-gdb vmlinux
```

Alternatively, start DDD with GDB:

```
ddd --debugger i386-linux-gdb vmlinux
```

4. At the gdb prompt (in the GDB command window within DDD), configure the remote connection by entering these two commands:

```
set remotebaud 9600
target remote /dev/ttyS0
```

Note: Some agents have communication routines that time out after lack of communication for a period of time. Because of this it is best to put GDB into “target remote” mode within about 30 seconds of the time the agent is started.

5. To start the debug agent,

 5a. Boot the target.

 5b. Log in as root.

 5c. Enter the following command:

 startkgdb -s <baudrate> -t <*serial port*>

 For example:

```
startkgdb -s 9600 -t /dev/ttyS0
```

You can now debug the target kernel from the host system. For more information about debuggers and how to use them, see “Additional Resources” on page 113.

Kernel Module Debugging

Complete the following steps to start a debugging session for the kernel:

1. In the Target Wizard tree view,

1a. Enable as modules any needed device drivers.

1b. Enable **Embedix > Kernel > Kernel hacking > Remote (serial) kernel debugging with GDB**.

1c. Enable **Embedix > Programming > Debugger > startkgdb**.

1d. Enable E**mbedix > Kernel > Enable loadable module support > Kernel module loader (CONFIG_KMOD)**.

1e. Enable appropriate serial drivers in the kernel.

Note: Make note of the path to the module to be debugged. (This path is relative to the Linux source directory where you will start the debugger.)

2. Deploy and boot the target image.

   For more information, refer to the "Building and Deploying a Target Image" section of your Embedix Target Wizard User Guide or refer to your board support package documentation.

Note: By default a stripped kernel has been placed in your target image and an unstripped kernel has been left in the project kernel source directory for use with GDB.

3. Connect a serial cable from the host system to the target system.
4. Ensure that the module is in the correct location on the target, and prepare the kernel to find the module.

   4a. Move a copy of modules to be debugged to `/lib/modules/`uname -r`/`<*module_category*>

   4b. Enter this command: **`depmod -a`**
5. On the host system, open an X-term and then complete the following steps at the prompt:

   5a. Provide paths to the executables by entering this command:

```
export PATH=/opt/Embedix/tools/bin/:/opt/Embedix/usr/
    X11R6/bin/:$PATH
```

5b. Go to the kernel source directory.

This is under the project directory you created with Embedix Target Wizard. The command will be similar to this:

```
cd /home/<username>/project/<projectname>
    /src/linux/
```

Note: In order for GDB to use the symbol information, the program's source code must remain in the same directory path it was in during compilation.

5c. Start the GDB executable for your target processor.

For example, to start GDB, enter this command:

```
i386-linux-gdb vmlinux
```

Alternatively, start DDD with GDB:

```
ddd --debugger i386-linux-gdb vmlinux
```

6. At the gdb prompt (in the GDB command window within DDD), configure the remote connection by entering these two commands:

```
set remotebaud 9600
target remote /dev/ttyS0
```

Note: Some agents have communication routines that time out after lack of communication for a period of time. Because of this it is best to put GDB into “target remote” mode within about 30 seconds of the time the agent is started.

7. To start the debug agent,

7a. Boot the target.

7b. Log in as root.

7c. Enter the following command:

`startkgdb -s <baudrate> -t` <*serial port*>

For example:

```
startkgdb -s 9600 -t /dev/ttyS0
```

You can now debug the target kernel from the host system. For more information on debuggers and how to use them, see “Additional Resources” on page 113.

8. At the `gdb` prompt, load the module for debugging. The path is relative to the kernel source path from where GDB was started:

`loadmodule` <*module_path*>**/**<*module_name*>**.o**

The module symbols are now available to breakpoint and debug as part of the kernel.

Note: If you are using a COM port other than ttyS0, you must update the following entry in the file src/linux/.gdbinit (replacing ttyS0 with your COM Port):

```
shell ./Documentation/kgdb/gdb_loadmodule $arg0
      /dev/ttyS0 > .gdbtmp
```

You can now debug the target kernel from the host system. For more information about debuggers and how to use them, see “Additional Resources” on page 113.

Application Debugging over IP

Complete the following steps to start a debugging session for debugging an application over IP:

1. In the Target Wizard tree view, complete these steps:

   1a. Enable appropriate network drivers in the kernel.

   1b. Enable **Embedix > Programming > Debugger > gdbserver**.

2. Prepare your custom application for debugging.

2a. Compile the application program with the -g flag to provide debugger symbols.

2b. Ensure that the original unstripped application remains in the project source directory for use with GDB.

2c. Place a stripped copy of the application program in the target image before deployment.

For detailed information, see Chapter 1, “Introduction to SDK Tools.”

3. Deploy and boot the target image.

   For more information, refer to the “Building and Deploying a Target Image” section of your Embedix Target Wizard User Guide or refer to your board support package documentation.

4. Connect host and target systems to the same network.

5. On the host system, open an X-term and then complete the following steps at the prompt:

5a. Provide paths to the executables by entering this:

```
export PATH=/opt/Embedix/tools/bin/:/opt/Embedix/usr/
    X11R6/bin/:$PATH
```

5b. Go to the application source directory.

If you added your custom application to your Embedix Target Wizard project, you will find your application in the following directory:

```
cd /home/<username>/project/<projectname>/src
    /<custom_application_source>/
```

Note: In order for GDB to use the symbol information, the program’s source code must remain in the same directory path it was in during compilation.

5c. Start the GDB executable for your target processor.

For example, to start GDB, enter this command:

```
i386-linux-gdb <application_name>
```

Alternatively, start DDD with GDB:

ddd --debugger i386-linux-gdb <*application_name*>

6. At the gdb prompt (in the GDB command window within DDD), configure the remote connection by entering this:

   target remote <*IP_address*>**:**<*Port_number*>

   Replace <*IP_address*> with the IP address of the target system. Use a valid (unused) port number between 1024-65535.

7. At the shell prompt on the target system, start the target debug agent by entering this:

   gdbserver <*IP_address*>**:**<*Port_number*> <*application-command-line*>

   Replace <*IP address*> with the IP address of the host system and be sure to use the same port number used in the previous step. Replace <*application-command-line*> with the desired application and arguments.

Note: Some agents have communication routines that time out after lack of communication for a period of time. Because of this it is best to put GDB into "target remote" mode within about 30 seconds of the time the agent is started.

You can now debug your custom application from the host system. For more information about debuggers and how to use them, see "Additional Resources" on page 113.

Application Debugging over Serial Port

Complete the following steps to start a debugging session for debugging an application over IP:

1. In the Target Wizard tree view, complete these steps:

   1a. Enable appropriate serial drivers in the kernel.

   1b. Enable **Embedix > Programming > Debugger > gdbserver**.

2. Prepare your custom application for debugging.

   2a. Compile the application program with the -g flag to provide debugger symbols.

   2b. Ensure that the original unstripped application remains in the project source directory for use with GDB.

   2c. Place a stripped copy of the application program in the target image before deployment.

   For detailed information, see Chapter 1, "Introduction to SDK Tools."

3. Deploy and boot the target image

   For more information, refer to the "Building and Deploying a Target Image" section of your Embedix Target Wizard User Guide or refer to your board support package documentation.

4. Connect a serial cable from the host system to the target system.

5. On the host system, open an X-term and then complete the following steps at the prompt:

   5a. Provide paths to the executables by entering this:

```
export PATH=/opt/Embedix/tools/bin/:/opt/Embedix/usr/
    X11R6/bin/:$PATH
```

   5b. Go to the application source directory.

   If you added your custom application to your Embedix Target Wizard project, you will find your application in the following directory:

```
cd /home/<username>/project/<projectname>/src
    /<custom_application_source>/
```

Note: In order for GDB to use the symbol information, the program's source code must remain in the same directory path it was in during compilation.

   5c. Start the GDB executable for your target processor.

For example, to start GDB, enter this command:

```
i386-linux-gdb <application_name>
```

Alternatively, start DDD with GDB:

```
ddd --debugger i386-linux-gdb <application_name>
```

6. At the gdb prompt (in the GDB command window within DDD), configure the remote connection by entering these two commands:

```
set remotebaud 9600
target remote /dev/ttyS0
```

7. At the shell prompt on the target system, start the target debug agent by entering this:

```
gdbserver /dev/ttyS0 <application-command-line>
```

   Replace <*application-command-line*> with the desired application and arguments.

Note: Some agents have communication routines that time out after lack of communication for a period of time. Because of this it is best to put GDB into “target remote” mode within about 30 seconds of the time the agent is started.

You can now debug your custom application from the host system. For more information about debuggers and how to use them, see “Additional Resources” on page 113.

Additional Resources

These tutorials were provided specifically to show how to start a debugging session in a way many software engineers may not have previously encountered. To learn how to perform debugging with GDB and DDD, see the following references:

- *GDB User's Manual*. A copy can be found on the Embedix SDK CD-ROM (browse the CD-ROM to index.html/documents/ Embedix Packages Technical References/gdb).

- *DDD User's Manual*. This can also be found on the Embedix SDK CD-ROM (browse the CD-ROM to index.html/documents/ Embedix Packages Technical References/ddd).
- Additional information about gdbstubs and kernel stubs are found in the Linux kernel source tree under linux/ Documentation/gdb-serial.txt.

CHAPTER 5 Helper Utilities

This section discusses utilities available to SDK users that are external to Embedix Target Wizard. Topics covered include:

“Building” on page 115
“Privileges” on page 116
“Deployment” on page 116

Building

Embedix Target Wizard uses a variety of external utilities to facilitate the build process, and `Builder.pm` is central to this whole system. It is a perl module that defines a class for building packages. `Builder.pm` itself is architecture-neutral with respect to its knowledge of building packages. However, for each architecture that Embedix Linux supports, there will be a subclass of `Builder.pm` which will be found in */opt/Embedix/emb-bin*.

There are two important scripts that make use of `Builder.pm` and its subclasses. The first is `emb_mkproj`, which is called whenever you start a new project or change to a different project within Embedix Target Wizard. Its purpose is to create the appropriate directories and symlinks inside the specified project directory so that the ECD information for Embedix Target Wizard can be initialized properly and that the correct toolchain for building is used.

The other script that uses `Builder.pm` is `emb_build`. Its purpose is to build a package for an Embedix Linux distribution. Embedix Target Wizard invokes this script once for each package during the build sequence. Together, these two scripts comprise the bulk of the build system.

Privileges

Another important helper utility is suwrapper. The purpose of suwrapper is to provide fine-grained root privileges for certain programs for certain users. The privilege rules are defined in */opt/Embedix/etc/suwrapper.conf*. If you are familiar with sudo, you will find suwrapper similar in concept. Without this program, neither emb_mkproj nor emb_build would run properly unless you were root. On the same note, if permission problems arise while trying to use Embedix Target Wizard, suwrapper may not be configured properly.

Deployment

emb_deploy is a Perl/TK script that facilitates the deployment of an Embedix Linux target image. This script was intended to be launched as a subprocess of Embedix Target Wizard, but it can be used as a stand-alone application as well.

The man pages that are relevant to the previously mentioned programs have been included here:

"emb_build" on page 117
"emb_mkproj" on page 118
"suwrapper" on page 119
"suwrapper.conf" on page 120
"tcconfig" on page 122

emb_build

NAME

emb_build -- build a package for an Embedix distribution.

SYNOPSIS

Usage

emb_build [OPTION]...

DESCRIPTION

emb_build is a script that builds packages for an Embedix distribution. It is a front-end to Builder.pm which provides a framework for cross-compiling packages for a wide array of architectures.

OPTIONS

emb_build requires that the following options be specified:

--projectdir DIR

DIR should be the path to where the project files are stored. This is usually $HOME/project.

--force

This forces a package to be built even if it theoretically doesn't need to be.

FILES

$PRJ_BASE/buildmsg

This is a named pipe that STDOUT and STDERR get redirected to so that external programs (such as Target Wizard) can monitor the progress of a build.

$PRJ_BASE is currently defined as the string passed to --spec modified by the regex, s-/build/rpmdir/SPECS.*--;

emb_mkproj

NAME

emb_mkproj - create a new Embedix Target Wizard project.

SYNOPSIS

Usage

emb_mkproj [OPTION]...

DESCRIPTION

The purpose of emb_mkproj is to create a new Embedix Target Wizard project. This creates a project directory if necessary and populates that directory with the data needed for configuring and building an Embedix distribution.

OPTIONS

emb_mkproj requires that the following parameters be specified:

board BOARD

BOARD should be the name of a perl module that is a subclass of Defaults that distills the architecture-specific information needed for cross-compilation.

projectdir DIR

DIR should be the path to where the project files will be stored. The specified directory will be created if it doesn't exist. By convention, Target Wizard defaults to using paths of this form:

$HOME/project/

uid UID

UID should be the user id of the current user.

gid GID

GID should be the group id of the current user.

suwrapper

NAME

suwrapper - fine-grained root privilege provider

SYNOPSIS

Usage

```
suwrapper <COMMAND> [ARG]...
Example
    $ suwrapper emb_build   \
        --spec=glibc.spec   \
        --pkg=glibc         \
        --board=mpc8260adsp \
        --srpm=glibc-2.2.1-2.srpm
```

DESCRIPTION

The purpose of suwrapper is to provide fine-grained root privileges. It is fine grained in the sense that the set of executables and the set of users that may have root privileges is specified in a configuration file called /opt/Embedix/etc/suwrapper.conf. Those of you who are familiar with sudo(8) will find suwrapper to be familiar in concept.

suwrapper itself is suid, so it is promoted to root privileges automatically by Linux when it is run. It reads the file /opt/Embedix/etc/suwrapper.conf, which contains a list of commands and the users who can execute them. The list of commands should ALWAYS be specified with absolute paths.

If a relative path is specified as the command to run, then

1. The PATH is searched.
2. The first file that matches has its fullpath expanded and is used to see if there is a match.
3. If the requested program is found in the configuration file, and the user who originally called suwrapper is listed, then the command is executed.

Note that suwrapper also automatically performs a chroot to /opt/Embedix for certain hardcoded programs. Specifically, it will do so for fakeroot and emb_build. In order to make sure that the original user can run inside the chroot, /etc/passwd and /etc/group are copied to /opt/Embedix/etc before chroot-ing.

Finally, if the command to be executed is fakeroot, privileges are returned to those of the original calling user (since fakeroot will take care of spoofing certain operations that require root privileges).

FILES

/opt/Embedix/etc/suwrapper.conf

This is the primary configuration file for suwrapper. For more information, see suwrapper.conf(8).

/etc/passwd

This file is used to identify users. It is also copied into /opt/Embedix/etc to facilitate chroot-ing.

/etc/group

This file is used to identify groups. It is also copied into /opt/Embedix/etc to facilitate chroot-ing.

suwrapper.conf

NAME

suwrapper.conf - configuration for suwrapper

DESCRIPTION

suwrapper.conf is the configuration file for suwrapper. It is expected to reside in /opt/Embedix/etc. It is very important that suwrapper be properly configured, because many key Target Wizard operations require privileged write permissions to work properly.

FORMAT

Comments

Lines beginning with '#' are ignored.

Commands and who can execute them

The first token is a command. It is recommended that the full path be given to minimize the risk of executing a trojan. All subsequent tokens are login names of those who are authorized to execute the said command.

EXAMPLE

```
# A sample suwrapper configuration file.  Suwrapper is used to
run some
```

```
# programs with root user permissions, these programs are listed
here along

# with the users who are allowed to run the command.

#

# Lines beginning with a # are ignored as are empty lines.

#

# Format is as follows:

#

# command user1 user2 user3 ...

/opt/Embedix/emb-bin/emb_build     knox root seh mattw tbird seth
beppu

/opt/Embedix/emb-bin/emb_mkproj    knox root seh mattw tbird seth
beppu

/opt/Embedix/usr/local/bin/fakeroot knox root seh mattw tbird
seth beppu

/opt/Embedix/bin/mount            knox root seh mattw tbird seth
beppu

/opt/Embedix/bin/umount           knox root seh mattw tbird seth
beppu

/opt/Embedix/bin/tar              knox root seh mattw tbird seth
beppu

/opt/Embedix/bin/cp        knox root seh mattw tbird seth beppu


# DO NOT DO THIS AS IT IS A SECURITY RISK:

#mount                                 knox
```

tcconfig

NAME

tcconfig - toolchain configuration for the Embedix SDK

DESCRIPTION

tcconfig files describe the various characteristics of a toolchain for a given architecture. The format of these files is line-based, where each line is of this form:

VARIABLE=value

The build system uses the information found in these files to properly invoke the appropriate cross-compiler for a given project.

VARIABLES

The following is a description of the variables found in a tcconfig file. Most of these are self-explanatory.

TC_TARGET_PREFIX

This is the prefix that is prepended to the cross-compiler (such as i386-linux-gcc)

TC_KERNEL_IMAGE_NAME

This is the name of the kernel image.

TC_KERNEL_ARCH

This is the architecture of the kernel.

TC_RPM_ARCH

This is the architecture of the RPM executable.

TC_ARCH_DESCRIPTION

This is a string used by Target Wizard to refer to this toolchain.

TC_GCC_INCLUDE_PATH
TC_GCC_LIBRARY_PATH

These are paths that gcc uses to compile and link with the appropriate files for a given architecture.

TC_SIZE_SHORT
TC_SIZE_INT
TC_SIZE_LONG
TC_SIZE_FLOAT

TC_SIZE_DOUBLE

These are architecture-specific data sizes.

TC_BOARD

This is the name of the board that this particular toolchain config applies to.

EXAMPLE

/opt/Embedix/tools/i386.tcconfig

```
TC_TARGET_PREFIX=i386-linux
  TC_KERNEL_IMAGE_NAME=bzImage
  TC_KERNEL_ARCH=i386
  TC_RPM_ARCH=i386
  TC_ARCH_DESCRIPTION=GNU/Intel 386
  TC_GCC_INCLUDE_PATH=/opt/Embedix/tools/lib/
 gcc-lib/i386-linux/2.95.2/include:/opt/
 Embedix/tools/lib/gcc-lib/i386-linux/2.95.2/
 ../../../../i386-linux/include:/opt/Embedix/
 tools/lib/gcc-lib/i386-linux/2.95.2/../../
 ../../include/g++-3
  TC_GCC_LIBRARY_PATH=/opt/Embedix/tools/lib/
 gcc-lib
  TC_SIZE_SHORT=2
  TC_SIZE_INT=4
  TC_SIZE_LONG=4
  TC_SIZE_FLOAT=4
  TC_SIZE_DOUBLE=8
  TC_BOARD=i386_default-2.0
```

The tcconfig files are expected to be in the /opt/Embedix/tools directory.

SEE ALSO

Builder(3pm), &Builder::parse_twcfg

APPENDIX

Manual Method of Packaging

This section provides a conceptual overview to manually creating the essential components of an Embedix package. We recommend you review this section to gain a fundamental understanding of packaging concepts, then refer to “Packaging with Package Editor” on page 5 and use Package Editor to create and edit your custom packages.

This section covers the following topics:

“What is an Embedix Package?” on page 125
“Creating an LBC” on page 126
“Creating an ECD” on page 136
“Using a Tarfile, SRPM, or CVS Directory For Source” on page 151
“Creating an SRPM” on page 153

What is an Embedix Package?

An Embedix package consists of at least three files:

- An LBC (Lineo Build Control) file containing the build control instructions for a package
- An ECD (Embedix Component Description) file containing information relevant to Target Wizard
- A source file (which could be a tarfile, SRPM file, or CVS directory) containing the source code and spec file of the package

Note: A package is permitted to have one or more patch files that will be applied after the SRPM or tarfile has been unpacked and prepared. These patches are activated by the presence of the %patches section in the LBC file.

Creating an LBC

The Embedix SDK builder software provides default build instructions for all packages. These defaults are contained in the Builder.pm file and should not be modified. They can, however, be overridden with entries in buildcontrol files.

A Lineo buildcontrol (LBC) file is used to provide information and build instructions to the SDK build engine. An LBC file is required for every software package.

Buildcontrol File Features

An LBC file has two important features:

- Granular “build” sections. In an LBC file, the build section is divided into the **%makec** and **%makeb** sections. This means that for most builds, a lengthy configure and rebuild-from-scratch step can be avoided.
- Independent sections. Each section provides either data or a shell script fragment. These sections are used at appropriate times in the build process.

Buildcontrol File Sections

The following table lists all the the sections available for use in an LBC file. Some sections are required, but most are not. Default build instructions exists for all SDK packages, so most sections are added to an LBC file only as needed to override a default.

Section	Purpose
%pkg_file	Source name (for example, linux-2.4.2-1.src.rpm). Required for every package.

%patches	Package specific patches. (The Builder automatically determines the patch level. Note that these patches are applied after any patches called out by the specfile when using an SRPM as the package file.)
%bld_dir_name	Build directory name. (This is required only if the directory created by the SRPM/tar file does not correlate with that of the package file name.)
%cflags	Compile flags applied to the CFLAGS of a package
%cfgopts	Extra options for configuration stage (for example, this is used with glibc)
%spec	Spec file name (This is only needed if the name of the spec file within an SRPM is not correlated to the SRPM name.)
%bin	Linux kernel name, including any subdirectory prefixes (for example, on x86 it may be arch/i386/boot/bzImage). Applies to the kernel only.
%bld_targ	The specific name of the make target required to build the Linux kernel for this platform (zImage, vimage, bzImage, linuz). Applies to the kernel only.
%makep	Instructions for the package prep stage
%makec	Instructions for doing one-time configuration
%makerc	Instructions for doing a repeat of the dynamic configuration of the package
%makeb	Instructions for compiling the source in the package
%makei	Instructions for building the virtual rooted target package (similar to the install section within an SRPM)
%makest	Instructions to stage components in the dev_image. dev_image is the location of libraries and other files needed to compile on a host system for a target. For instance, you may require a library from one package in order to build another, but the library itself is not required on the target.

%makedc	Instructions for removing object code and other generated material, and to place in a state ready to be configured and built again.

Buildcontrol Inheritance

Default build instructions that are contained in the Builder.pm file should not be modified. They can, however, be overridden at any level as needed with sections and section entries in LBC files.

The builder looks for LBC files in three places: the generic, board, and local directories, which are located in the directory:

<*project_name*>/config-data/buildcontrol/

The **generic** directory contains package-dependent LBC files and sections that are valid for any board or project.

The **board** directory contains architecture-dependent LBC files and sections that override above behaviors and fix very specific architecure build problems.

The **local** directory contains local project development LBC files and sections that override any of the above sections. Personal software development should be specified in this section.

Sections in LBC files from each of these locations are merged, giving priority first to local, then to board, and then generic.

End-user LBC files would normally be placed in the local subdirectory and would have precedence over all other LBC files.

Example 1:

The following example shows the masking behavior for a project that has LBC files with section entries in the four locations described. In this example, the sections included in each LBC file are represented with letters.

A = %makep
B = %makec
C = %makeb
D = %makei

E = %makedc

```
Location      LBC File Sections
builder.pm    A B C D E
generic       A - - D E
board         - - C - E
local         A - - - -
              | | | | E from board/<pkgname>.lbc
              | | | D from generic/<pkgname>.lbc
              | | C from board/<pkgname>.lbc
              | B from Builder.pm (SDK default)
              A from local/<pkgname>.lbc
```

The point of this example is to demonstrate that each of the files can have identical sections, but the behavior is dependant on a specific hierarchy. If you want to override a certain section, then you only need to include that section (plus the only required section %pkg_file) in the LBC file. In other words, you do not need a complete LBC file in order to override one section.

Example 2:

- /generic/bash.lbc has the sections %makei, %bin, %pkg_file, and %makeb
- /board/bash.lbc has two sections, %makedc and %makei
- /local/bash.lbc has two sections, %makeb and %makei

Here, the builder would use the local version of %makeb and %makei, the board version of %makedc, and the generic version of all the rest.

This way, if there is a global change to the %bin section for bash, you only need to change the %bin in the generic directory and everything will still build safely for the different boards because all projects will “inherit” the change.

Build Phases

For most packages the first time a package is built the following sections will be executed in this order. (The build sequence for subsequent builds will be determined by the Package build steps.)

%makep
%makec
%makeb
%makei
%makest

%pkg_file

The name of the file containing the source materials for this package. This is the only section of the LBC file that is absolutely required. Other sections may be optional or may utilize default behavior provided by the build engine itself.

This name is matched against several patterns to determine the appropriate action for certain build phases (particularly the "install source" or "prep" phase).

The following patterns are detected:

tar.gz, .tgz, or .bz2 = Indicates a compressed archive of a source tree

.src.rpm = Indicates a source RPM file (SRPM) with its own source archive, specfile, and optional patches

CVSROOT = Indicates a source repository

%patches

This indicates a list of patch filenames for patches that are applied at the end of the "install source" or "prep" phase. These are source

patches that are applied to the source tree after it has been installed during that phase (in the build directory for this package).

The patches themselves reside in one of the subdirectories of <project/name>/Packages (generic, board, or local).

%bld_dir_name

This is the name of the directory where the source materials reside, in the project directory under <project/name>/build/rpmdir/BUILD/bld_dir_name.

%cflags

This section indicates compilation flags that are appended to the normal CFLAGS used by a package. These flags are put into the environment variable OPT_FLAGS before the build of a particular package. They override the project CFLAGS during the package compilation. The default CFLAGS can be found in the following file: (project)/config-data/buildcontrol/board/bsp_config

%cfgopts

This section has extra options used for the configure script. The glibc package configuration step can be customized from the command line. The extra options can be passed in by including them in this section.

%spec

The spec file name doesn't match the short name on some packages. (Currently glibc is the only package that falls into this category.) This allows the system to explicitely name the spec file for an RPM build.

%bin

The name and/or location of the kernel at the end of the build cycle.

%bld_targ

This section indicates the "build target" name, or the name of the resulting kernel image after a kernel build. This section is only used

for the kernel. The actual kernel image name varies depending on the architecture and deployment system used and can vary from bsp to bsp.

Some example names are:

zImage
vimage
bzImage
vmlinuz

%makep

These are the instructions for the "prep" phase, which is the phase during which the source materials for a package are installed into the project build area. The source materials are located in one of the directories: <project/name>/Packages/{generic, board, local} and are specified by the %pkg_file and %patches sections.

The result of performing this stage is that the package source materials are located under <project/name>/build/rpmdir/BUILD directory and are ready to build.

If this section is missing, then the default behavior depends on the package type. The package type is determined by matching the pkg_file name to one of the patterns mentioned above. This can be archive, SRPM, or CVS.

For an SRPM, the default behavior is to install the software using the "rpm -bp" command. This means that the SRPM is installed, placing the SRPM archive and patches into <project/name>/build/rpmdir/SOURCES, and the specfile into build/rpmdir/SPECS. Then the software is placed into <project/name>/build/rpmdir/BUILD/<bld_dir_name>.

For an archive, the default behavior is to untar the software directly into <project/name>/build/rpmdir/BUILD.

For a CVS repository, the behavior is to check out the software into <project/name>/build/rpmdir/BUILD.

No matter what the package type, any patches that are specified are then applied to the resulting source tree. This means that end-user

patches can be applied on top of source that has already been patched by the RPM build-prep operation.

Note that the patches reside in one of the Packages subdirectories (generic, board, or local), just like other source package formats. Also, the patches are applied in a way that auto-determines the appropriate patch level based on information in the patch and in the source tree.

After this phase is complete, the package state file is created (<project/name>/build/packagestate/<pkg>.st) and a "bp" entry is made in the file. This indicates that the software source has been "prepped" or installed in the build area of the project.

If you need to re-install the software (in order to return it to a pristine state), you can remove the package state file and build the package from Target Wizard. This will cause the builder to reprep the software. The package state files are located in the <*project_name*>/build/packagestate/"package".{bi,st}.

If the builder tries to reprep the software and a directory already exists for it under the BUILD directory, then the installed directory is moved from <*project_name*>/build/rpmdir/BUILD/<*bld_dir_name*> to <*project_name*>/build/rpmdir/BUILD/<*bld_dir_name*>.modified. This is done to preserve any changes that the user may have made to the source.

If <*bld_dir_name*>.modified already exists, then the builder stops and displays an error message, allowing the user to correct the situation. You may remove or rename the .modified directory.

%makec

These are the instructions used to configure the software for building. This usually consists of running a configuration script, which auto-detects many aspects of the host system, including tools and libraries. This step usually reverts the source tree into a state where a full recompile is required. Hence it has been split out into its own phase (separate from the "build" phase).

After this phase is run, an entry "cfg" is made to the package state file. You can force a package to run through its configure phase by

removing this entry from the package state file found in the following location: <*project_name*>/build/packagestate/<*pkg_name*>.st.

The default action in the Builder.pm file for the phase is "makeconfigure."

%makerc

This is only used by the kernel and is used to configure the kernel after the first configuration has already been run. This saves time, because certain operations do not need to be run every time you reconfigure the kernel.

%makeb

These are the actual instructions used to build the software from the source code. At the end of this phase, the software binaries have been created, but they still reside inside the source tree under <project/name>/build/rpmdir/BUILD/<bld_dir_name>.

These instructions are run for a normal "build" operation and should be able to remake the software if it has been changed. In particular, it should support incremental rebuild so that only the modified or unbuilt files are compiled (or recompiled).

The default behavior for this stage (if this section is missing) is simply "make," at the root of the <bld_dir_name>.

%makei

These instructions are used to install the binaries from the software (usually to a "fake" root area) where they can be collected for placement into a tarfile.

If this section is empty, then packages of type SRPM result in a call to "rpm -bi." This means that the instructions in the package SPECFILE are used to install the software, and the %Files section of the SPECFILE is used to populate the contents of ultimate tarfile.

After this step is complete, a tarfile for this package should be found in the directory <project>/build/tarfiles. If there is a link in that

directory (back to the bsp directory where the prebuilt binaries are), then that link is replaced with the real file that results from the build.

%makest

This phase is used to "stage" build materials into the dev_image area <project/name>/build/dev_image. The dev_image is an area where include files, libraries, and other development materials are placed. Other packages may reference these files in order to build. By referencing these files, the target enviroment will not reference host libraries and include files.

The %makest typically looks something like this:

```
tar zxf %TARDIR/%PKG.tar.gz ./ursr/lib ./usr/include
```

This will extract the files in /usr/lib, include and install them in the dev_image directory.

%makedc

Commands listed in this phase can be used to restore a source tree to a completely fresh state. If a "Makefile" system is used, a "make clean" would be the default behavior. This step does not remove user modifications. This phase is called before a forced rebuild of a package and should result in all other build phases being run, except for the prep phase.

Example

The following is a simple hello.lbc file.

```
# hello_world build controls

%pkg_file
hello_world-1.0.tgz

%bld_dir_name
hello_world-1.0

%makedc
make clean

%makep

%makep
make
```

```
%makei
if [-e $BUILD_ROOT]
then rm -rf $BUILD_ROOT
fi
mkdir -p $BUILD_ROOT/usr/bin
CP $BUILD_DIR/hello_world $BUILD_ROOT/usr/bin
```

Creating an ECD

The Embedix Component Description (ECD) file contains information that Target Wizard uses to provide descriptions and options to build packages. Many packages require simple ECD files that can be written with a few minutes' worth of work.

ECD format is similar to nested tags in HTML. To guide you through the steps involved in creating a typical ECD file, we use a sample application Hello World. The examples presented in this sample ECD should be sufficient for most applications.

Note: For further examples of ECDs, review the ECDs in /opt/Embedix/embedix-2.0/config-data/ecds on your build platform (the development host machine).

A Typical ECD File

The following example is an ECD that begins with a comment (a line beginning with a #) on line 1:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
-------------------
```

Where this package appears in the component tree depends on which group is specified. If you don't specify a group, then the package will appear under the root Embedix component. Consider

creating a new group for your package named "custom," as follows on lines 2 and 3:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
</GROUP>
-------------------
```

Each package is contained in a component description with a unique name:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
</COMPONENT>
</GROUP>
-------------------
```

The SRPM file and the specpatch are also specified. These are legacy fields since the source code and patches are specified in the .lbc file. Specpatch is not normally used unless you are adding an SRPM. Only the base of the SRPM file needs to be named.

For example, if your SRPM were named hello_world-1.0-1.src.rpm, you would need to enter "SRPM=hello_world" only.

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
SRPM=hello_world
```

```
<SPECPATCH></SPECPATCH>
</COMPONENT>
</GROUP>
-------------------
```

Next, add a help message for this component:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
SRPM=hello_world
<SPECPATCH></SPECPATCH>
<HELP>This is my custom application.
It's good!
</HELP>
</COMPONENT>
</GROUP>
-------------------
```

This help message appears in the Description tab in Target Wizard when the component or node is selected. (This particular help section can have multiple lines; you can follow this example if you need a longer message.)

Now add some base information for the component:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
SRPM=hello_world
```

```
<SPECPATCH></SPECPATCH>
<HELP>This is my custom application.
It's good!
</HELP>
TYPE=bool
DEFAULT_VALUE=1
PROMPT=Include custom application hello_world?
</COMPONENT>
</GROUP>
------------------
```

The type field is assigned a boolean value, so that you have two choices (include hello_world or exclude hello_world). The default_value field will be 1, because hello_world will probably be included most of the time. The prompt field contains a string that will appear in the Node list in Target Wizard.

Now think about what this component provides and what it requires for and from other packages. In this case, this component provides the application hello_world. Suppose that it is dynamically linked to libc.so, but to no other libraries (which you could determine by using the ldd command).

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
     SRPM=hello_world
     <SPECPATCH></SPECPATCH>
     <HELP>This is my custom application.
           It's good!
     </HELP>
     TYPE=bool
```

```
    DEFAULT_VALUE=1
    PROMPT=Include custom application hello_world?
    <PROVIDES>
       hello_world
    </PROVIDES>
    <REQUIRES>
       libc.so.6
    </REQUIRES>
</COMPONENT>
</GROUP>
------------------
```

Now any other package that requires hello_world can specify hello_world in its REQUIRES field. Similarly, the package providing libc.so must have libc.so.6 in its PROVIDES field (which it does).

To tell Target Wizard which files are needed for application hello_world, a KEEPLIST option is used. If you want all files, include this line:

KEEPLIST=/

Suppose that your application installs the files /usr/bin/hello_world, /usr/local/share/hello_world/hello_world.cfg, and /usr/man/man1/hello_world.1. If you want the executable and the configuration file on an embedded target but not the documentation (the man page), your ECD should be as follows:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
SRPM=hello_world
<SPECPATCH></SPECPATCH>
<HELP>This is my custom application.
It's good!
```

```
</HELP>
TYPE=bool
DEFAULT_VALUE=1
PROMPT=Include custom application hello_world?
<PROVIDES>
hello_world
</PROVIDES>
<REQUIRES>
libc.so.6
</REQUIRES>
<KEEPLIST>
/usr/bin/hello_world
/usr/local/share/hello_world/hello_world.cfg
</KEEPLIST>
</COMPONENT>
</GROUP>
------------------
```

Note: You don't need to include directories in the keeplist, only the specific files. If you do enter a directory in the keeplist, then Target Wizard is instructed to place that entire directory tree on the target.

Finally, some miscellaneous options can be included (omitting them will not break anything).

STATIC_SIZE contains the amount of memory used by this component. Use the command **`size`** ***<objfile>*** to determine size of every item in your keeplist. Use the decimal value output. (For example, the command **`size hello_world`** might display a “dec” size of 1343.) Don’t forget to add up all files in the keeplist.

STORAGE_SIZE is the size of all the files in the keeplist of this component. They can be found simply by using the ls command with the -l option.

STARTUP_TIME is the time (in seconds) that is required to start this component for the first time. If you think this is significant and you know the value, specify it using this field. This is normally set to 0.

All these values are used for size and time calculations by Target Wizard. If they are not included, Target Wizard will assume the value is 0. Although this does not affect whether you can build, be aware that your size calculations will be off if these values are not correct.

For the example, assume that /usr/bin/hello_world requires 25,792 bytes on disk, and /usr/local/share/hello_world/hello_world.cfg requires 981 bytes, and hello_world's memory size is 15,607. Also assume that STARTUP_TIME is insignificant. The ECD example then becomes this:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
     SRPM=hello_world
     <SPECPATCH></SPECPATCH>
     <HELP>This is my custom application.
            It's good!
     </HELP>
     TYPE=bool
     DEFAULT_VALUE=1
     PROMPT=Include custom application
hello_world?
     <PROVIDES>
        hello_world
```

```
      </PROVIDES>
      <REQUIRES>
         libc.so.6
      </REQUIRES>
      <KEEPLIST>
         /usr/bin/hello_world
         /usr/local/share/hello_world/
hello_world.cfg
      </KEEPLIST>
      STATIC_SIZE=15607
     STORAGE_SIZE=26773
     STARTUP_TIME=0
</COMPONENT>
</GROUP>
------------------
```

Build Options in an ECD

Now you can explore using Target Wizard for more advanced uses of build options.

Build Option 1

The first type of build option allows you to include or exclude groups of files. For example, suppose application hello_world has a support application that provides online help called *hello_world-help*. You can provide this option simply by adding an OPTION section to the ECD:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
     SRPM=hello_world
```

```
        <SPECPATCH></SPECPATCH>
        <HELP>This is my custom application.
           It's good!
        </HELP>
        TYPE=bool
        DEFAULT_VALUE=1
        PROMPT=Include custom application
hello_world?
        <PROVIDES>
           hello_world
        </PROVIDES>
        <REQUIRES>
           libc.so.6
        </REQUIRES>
        <KEEPLIST>
           /usr/bin/hello_world
           /usr/local/share/hello_world/
hello_world.cfg
        </KEEPLIST>
        STATIC_SIZE=0
        STORAGE_SIZE=26773
        STARTUP_TIME=0
<OPTION include-hello_world-help>
        TYPE=bool
        DEFAULT_VALUE=1
        PROMPT=Include online help for hello_world?
       <HELP>If you have room for the online help,
you should probably include it. If you don't
include it, hello_world will still work, but the
user might need to consult a hard copy manual.
```

```
      </HELP>
      <KEEPLIST>
         /usr/local/share/hello_world/
hello_world.help
      </KEEPLIST>
      STATIC_SIZE=0
      STORAGE_SIZE=7548
      STARTUP_TIME=0
</OPTION>
</COMPONENT>
</GROUP>
------------------
```

Note: Each option must be enclosed in an OPTION tag, and all fields under the component field can also be under an OPTION field.

Build Option 2

The second kind of build option allows you to change configuration at build time.

For this example, suppose you want to include debugging symbols in the hello_world binary. Suppose too that your spec file has the following in the build section:

```
hello_world.spec:
-----------------------
%Build
%{DESTDIR}
CFLAGS="-O2";export CFLAGS
make
-----------------------
```

To enable debugging, you need to pass -g to the compiler in the CFLAGS environment variable. To do this, first create a placeholder for the option in a copy of the spec file:

```
hello_world.spec.new:
-----------------------
%Build
%{DESTDIR}
CFLAGS="-O2 !!HELLO_WORLD_CFLAGS!!";export
CFLAGS
make
-----------------------
```

Another method is to just add a %cflags section to the hello_world.lbc file. For example:

```
%cflags -g
```

Creating a Specpatch File

Specpatch files are needed only when configuration options need to be included in the RPM spec file. Otherwise, specpatch files can be omitted.

If a specpatch file is required, this is the point at which you need to create it. If hello_world.spec is the original and hello_world.spec.new contains your changes, you can create the specpatch using the command diff -c hello_world.spec hello_world.spec.new > hello_world.specpatch. You then copy hello_world.specpatch to /opt/Embedix/embedix-2.0/config-data/specpatches.

The !!HELLO_WORLD_CFLAGS!! in your new spec file is referred to as a *build variable*. You can change the value of HELLO_WORLD_CFLAGS by inserting another option in the ECD:

```
hello_world.ecd:
-------------------
#ECD for Hello World.
```

```
<GROUP Custom>
<COMPONENT hello_world>
     SRPM=hello_world
     <SPECPATCH>hello_world.specpatch</
SPECPATCH>
     <HELP>This is my custom application.
        It's good!
     </HELP>
     TYPE=bool
     DEFAULT_VALUE=1
     PROMPT=Include custom application
hello_world?
     <PROVIDES>
        hello_world
     </PROVIDES>
     <REQUIRES>
        libc.so.6
     </REQUIRES>
     <KEEPLIST>
        /usr/bin/hello_world
        /usr/local/share/hello_world/
hello_world.cfg
     </KEEPLIST>
     STATIC_SIZE=0
     STORAGE_SIZE=26773
     STARTUP_TIME=0
<OPTION include-hello_world-help>
     TYPE=bool
     DEFAULT_VALUE=1
```

```
     PROMPT=Include online help for hello_world?
    <HELP>If you have room for the online help,
you should probably include it. If you don't
include it hello_world will still work, but the
user might need to consult a hard copy manual.
     </HELP>
     <KEEPLIST>
        /usr/local/share/hello_world/
hello_world.help
     </KEEPLIST>
     STATIC_SIZE=0
     STORAGE_SIZE=7548
     STARTUP_TIME=0
</OPTION>
<OPTION hello_world-debug>
     TYPE=bool
     DEFAULT_VALUE=0
     PROMPT=Turn on debugging symbols for
hello_world?
     <HELP>
       You need these to follow what's going on
with a debugger.
     </HELP>
     <BUILD_VARS>
        HELLO_WORLD_CFLAGS="-g"
     </BUILD_VARS>
</OPTION>
</COMPONENT>
</GROUP>
------------------
```

Note that the SPECPATCH option contains hello_world.specpatch. Now you have an option from Target Wizard to specify a build with debugging symbols on.

Now suppose that you'd like to be able to specify more flags to the compiler. Then you can do the following:

```
hello_world.spec:
-----------------------
%Build
%{DESTDIR}
CFLAGS="-O2 !!HELLO_WORLD_CFLAGS!!";export
CFLAGS
make
-----------------------
hello_world.ecd:
-------------------
#ECD for Hello World.
<GROUP Custom>
<COMPONENT hello_world>
     SRPM=hello_world
     <SPECPATCH>hello_world.specpatch</
SPECPATCH>
     <HELP>This is my custom application.
        It's good!
     </HELP>
     TYPE=bool
     DEFAULT_VALUE=1
     PROMPT=Include custom application
hello_world?
     <PROVIDES>
        hello_world
```

```
        </PROVIDES>
        <REQUIRES>
           libc.so.6
        </REQUIRES>
        <KEEPLIST>
           /usr/bin/hello_world
           /usr/local/share/hello_world/
hello_world.cfg
        </KEEPLIST>
        STATIC_SIZE=0
        STORAGE_SIZE=26773
        STARTUP_TIME=0
<OPTION include-hello_world-help>
        TYPE=bool
        DEFAULT_VALUE=1
        PROMPT=Include online help for hello_world?
        <HELP>If you have room for the online help,
you should probably include it. If you don't
include it hello_world will still work, but the
user might need to consult a hard copy manual.
        </HELP>
        <KEEPLIST>
           /usr/local/share/hello_world/
hello_world.help
        </KEEPLIST>
        STATIC_SIZE=0
        STORAGE_SIZE=7548
        STARTUP_TIME=0
</OPTION>
<OPTION hello_world-cflags>
```

```
     TYPE=string
     DEFAULT_VALUE="-O2"
     PROMPT=Enter CFLAGS to control compile
options.
     <HELP>
Enter command line options to be passed to the
compiler:
     </HELP>
     <BUILD_VARS>
        HELLO_WORLD_CFLAGS="$VALUE"
     </BUILD_VARS>
</OPTION>
</COMPONENT>
</GROUP>
------------------
```

Summary

You can change the spec file of the package based on option values viewed under Target Wizard.

Any configuration item that an environment variable can contain can be placed in the Target Wizard Node list (and that should cover almost all possible configuration options). Use this method whenever you have a configuration that changes from target to target.

Using a Tarfile, SRPM, or CVS Directory For Source

The Embedix SDK currently supports three forms of package archives: tarfile, source RPM (SRPM), and CVS repository. The preferred form is tarfile.

The archive (whichever one you choose) is placed in the directory <*project*>/Packages/local.

Tarfile

Tarfiles are usually available for open source packages directly from the Internet. If you have obtained a source tree in another form, or if you are dealing with source that you have created yourself, then you will need to make the tarfile yourself from the source directory tree.

A convention that is used in the industry is to include the version number of the software in the tarfile name.

<*program_name*>-<*program_version*>.<*file_type*>

Notice that the program name and program version are separated by a dash. For example, if the code you were making a tarfile for was the "hello world" program, version 1.0, a good tarfile name would be:

hello-1.0.tgz

The extension in this case, is a gzip-compressed tarfile. The extension .tgz is an abbreviation for tar.gz. The extensions .tgz, .tar.gz, and .bz2 (bzip2 compressed tarfile) are all recognized by the SDK build engine.

You can create a tarfile for a source directory tree by going to the root of the tree and typing:

```
tar -czvf hello-1.0.tgz hello/*
```

The resulting archive "hello-1.0.tgz" should be placed in the <*project*>/Packages/local directory.

SRPM

The SRPM file can be placed directly in the <*project*>/Packages/local directory and, after creating the .lbc and .ecd files, the SRPM file should be usable.

CVS Repository

The CVS repository can be specified in the %pkg_file section of the .lbc file in the format:

```
CVSROOT=<directory>/package.
```

The SDK will check out a current copy of the source code into the build area and proceed with the build sequences.

Creating an SRPM

An SRPM file is generated by the RPM command. This is mostly for legacy purposes, since the SDK makes it very easy to add simple tar files using a .ecd, .lbc, and the source code. Building an SRPM involves these steps:

1. Creating a tar file of your source
2. Creating a spec file
3. Placing these in the correct location on your computer (for example, under /usr/src/OpenLinux if you are working on OpenLinux 2.4)
4. Running rpm -ba package.spec, where package is the name of your package

Note: For more information about the RPM suite and SRPM files, you can check <www.rpm.org>.

Index

A

adding configurations 5
adding custom applications 5

B

buildcontrol
  build sequence 130
  file sections 126

C

commands
  rpm 153
conventions used in manual v
creating ECDs 136
creating SRPMs 153
customizing a project 5

D

ddd 95
debugging overview 95
debugging tools 95
debugging tutorials
  Additonal resources 113
  Application debugging over IP 109
  Application debugging over serial port 111
  Kernel debugging at boot time 102
  Kernel debugging started from the shell 105
  Kernel module debugging 106
  Overview 101
documentation conventions v

E

ECD, example 136
ECDs, creating 136
Embedix tool chains 58

G

gdb 95

L

lbc file sections
  %bin 131
  %bld_targ 131
  %cfgopts 131
  %cflags 131
  %makeb 134
  %makec 133
  %makedc 135
  %makei 134
  %makep 132
  %makerc 134
  %makest 135
  %patches 130
  %pkg_file 130
  %spec 131
  bld_dir_name 131

lbc file sections, explained 126

N

notes v

P

package
  essential files 6, 125
project
  customize 5

S

SRPMs
  creating 153

T

tips v
tool chains 58

U

using tool chains 58

W

warnings v