



LINEO®

Embedix SDK 2.4.1 Tools Addendum

Contents

Embedix SDK Tools Addendum Overview	5
What's New?	5
VNC Remote Connection How-To	6
Software Description:	6
Installation	6
VNC Server	6
VNC Viewer	7
Using VNC for Remote Support	8
IT Department Information:	9
Summary:	9
Home Network Users:	9
Replacing the Target Wizard SPLASH Screen with a new Image	10
Debian Binary Import Tool (DBIT)	11
Importing a Debian Package	11
The import/export panel:	12
Modifying the ECD	13
Modifying the pre/post install scripts	14
Creating a New BSP	15
Step 1: Open a SDK Project of the Same Architecture	15
Step 2: Importing a New/Patched Kernel into the Project	15
Example: Modifying the Existing Embedix Kernel	16
Importing a New Kernel	16
Example: x86 Kernel Replacement	17
Example: Kernel Patch, Update and Replacement	17
Step 3: Import New Device Drivers	20
Device Driver Import Process	20
Makefile changes:	20
Create LBC and ECD Files with Package Editor	21
Kernel Module Example Source Code	22
Kernel Module Example Makefile	22

Contents

Step 4: Import a New GNU Tool Chain	23
GNU Tool Elements	23
GCC:	23
GDB:	24
Binutils:	24
Other Software:	24
Locating Updated GNU Tools	25
Gnu Tool Naming	25
New Tools Integration Steps	26
GCC Elements:	26
GCC from other Linux Sources	27
Building your own Tool Chain	28
Binutils:	29
Libraries:	29
Header Files	29
Modifying tc.config	29
Testing New Tool Chains	29
Step 5: Import New Applications and Services	30
Step 6: Modify the Deployment Wizard	30
Deployment Architecture	30
Important Files	30
Sequence of events when deploying	31
Deployment Phases	31
Writing a deployment wizard	32
Getting Started	32
Using the page library (pagelib.py)	32
Running shell commands from within the wizard	34
Running as Root	34
Utility Functions	35
Deployment Guidelines	35
Deployment Hooks	36

Contents

<i>Requirements</i>	36
<i>deploy_page</i>	36
<i>Validate(self,database)</i>	37
<i>save_config(self, database)</i>	37
<i>do_action(self, database, action)</i>	37
<i>do_package_sections(self, database, lbc_section)</i>	38
<i>Usage notes</i>	38
<i>Importing deploy_page</i>	38
<i>Calling deploy_setup(database)</i>	38
<i>The LDC File</i>	39
<i>Package Deploy Configuration</i>	40
<i>Files</i>	43
<i>Manual Deployment</i>	43
<i>Step 7: Creating a New BSP from the Project</i>	44
<i>Using export_bsp</i>	44
<i>Embedix Support</i>	45
<i>Notices</i>	47

Embedix SDK Tools Addendum Overview

This document describes additional features released within Embedix SDK v2.4.1-1 and is designed to supplement the existing documentation. There is little overlap between the original Embedix Tools Guide and this document.

What's New?

Embedix SDK v2.4.1-1 contains updates, bug-fixes and new enhancements. This list includes:

- **New!**
 - VNC Remote support capability software added and documented
 - Debian Binary Import Tool (DBIT)
 - LPF Mime Type for KDE
 - External SPLASH screen
 - Update Wizard v 2.4
 - dot-config Import Update
 - example files
 - ~/cdrom/examples
 - Enhanced documentation
 - Located on the CD at /documents and installed to the host at /opt/Embedix/documents.GNU documents
 - Advanced Linux Programming – by Mark Mitchell, Jeffrey Oldham, and Alex Samuel, of CodeSourcery LLC, published by New Riders Publishing, ISBN 0-7357-1043-0, First Edition, June 2001. Available under the Open Publication License, Version 1, no options exercised. (See: <http://www.advancedlinuxprogramming.com/about.html>)
 - Linux Device Drivers – by Alessandro Rubini & Jonathan Corbet, published by O'Reilly & Associates, ISBN 0-59600-008-1, 2nd Edition June 2001. Available under the GNU Free Documentation License. (See: <http://www.xml.com/ldd/chapter/book/index.html>)
 -
- **Updates**
 - Package Editor Update
 - Busy Box Update
 - DDD Update
 - new libstdc++ for x86

VNC Remote Connection How-To

To better assist our customers with the configuration and use of their tools, we have included VNC software on the Embedix SDK installation CD. This VNC software will allow the Embedix support staff to interactively control your own installation to speed troubleshooting and to better resolve training issues. Though this feature is not required, enabling this will allow the Embedix staff to support you more efficiently.

This chapter describes the configuration and use of Virtual Network Computing.

Software Description:

TightVNC - "VNC (an abbreviation for Virtual Network Computing) is a great client/server software package allowing remote network access to graphical desktops. With VNC, you can access your machine from everywhere provided that your machine is connected to the Internet. VNC is free (released under the GNU General Public License) and it's available on most platforms.

TightVNC, includes a lot of new features, improvements, optimizations and bug fixes over the original VNC version, see the list of features below. Note that TightVNC is absolutely free, cross-platform and compatible with the standard VNC. Many users agree that TightVNC is the most advanced free remote desktop package. And it's being actively developed so you can expect TightVNC will become even better.

TightVNC can be used to perform remote administration tasks in Windows, Unix and mixed network environments. It can be very helpful in distance learning and remote customer support. Finally, you can find a number of additional VNC-compatible utilities and packages that can extend the areas where TightVNC can be helpful.

TightVNC is a project maintained by Constantin Kaplinsky. Many other individuals and companies participate in development, testing and support." -- <http://www.tightvnc.com/index.html>

Website: <http://www.tightvnc.com>

Embedix CD software and document location: /misc

Installation

A VNC system consists of two components, a VNC Server and a VNC Viewer. The VNC server should be installed on the Development system running Embedix SDK. The VNC Viewer should be installed on the development system, and on any other system which desire to view the development system remotely.

VNC Server

The Embedix SDK CD includes RPM files for installation of the Tight VNC optimized server on Linux platforms. To install VNC Server, follow the procedure below:

-
- super-user (e.g. su)
 - mount the Embedix SDK CD (e.g. mount /mnt/cdrom)
 - Change directory to the VNC contents (e.g. cd /misc)
 - rpm -Uvh vnc-server-3.3.3r2+tight1.2.6-1.i386.rpm
 - Note: this is a [Red Hat](#) Linux (and compatible distributions) RPM file.
 - If you are using a non-Red Hat compatible distribution, we have provided the source code for your convenience.
 - Install KDE as the default desktop for the virtual sessions.
 - cp \$HOME/.vnc/xstartup \$HOME/.vnc/xstartup.orig
 - cp xstartup \$HOME/.vnc/xstartup
 - Install a new version of the vncserver script which is more distribution independent than the original (new font paths)
 - cp /usr/bin/vncserver /usr/bin/vncserver.orig
 - cp vncserver.new /usr/bin/vncserver

VNC Viewer

The directory /miscutils on the CD includes VNC viewers for both Windows (TM) and Linux platforms.

Q: Why install VNCViewer on the development platform?

A: VNCServer does not mirror the display which is currently shown on your monitor, but instead it creates a new and independent desktop which is only accessible via the network. VNCviewer is required to bring that network accessible display out to your physical monitor.

To install VNC viewer, follow the procedure below:

- super-user (e.g. su)
- mount the Embedix SDK CD (e.g. mount /mnt/cdrom)
- Change directory to the VNC contents (e.g. cd /miscutils)
- rpm -Uvh vnc-3.3.3r2+tight1.2.6-1.i386.rpm
 - This is a [Red Hat](#) Linux (and compatible distributions) RPM file.
 - If you are using a non-Red Hat compatible distribution, we have provided the source code for your convenience. Alternatively you can use RPMFind at <http://rpmfind.net/linux/RPM/> or <http://rpmfind.net/linux/RPM/Distrib.html> to locate a version for your distribution.

Note: A windows version is also available on the CD at /miscutils, for which installation is self explanatory:

Using VNC for Remote Support

1. Pre-configure the network as required and summarily described below.
2. Start the VNCserver on the development platform
 - o `vncserver -geometry 800x600 -depth 16`
 - `-geometry` specifies the screen resolutions. Standard values should be used: 640 x 480, 800x600, 1024x768
 - `-depth 16` specifies the color depth. Either 16 or 24 should be used. The use of 16 will reduce bandwidth requirements. Target Wizard requires a depth of at least 16.
 - o Note: Detailed vncserver help can be found by typing:
 - `man vncserver` : provides detailed information on usage.
 - `vncserver --help` : provides summary usage information
 - `vncserver --junk` (or any other non-recognized string) : provides more details on usage
 - o The first time that VNCserver is started, you are required to provide a password which must be entered by each VNCviewer client.
 - This password can be changed in the future by `/usr/bin/passwd`
3. Start the VNCviewer on the development system (required to monitor what the remote party is doing on your own machine)
 - o at the prompt, enter: **vncviewer**
 - o enter in the local IP address of that development system (found by `/sbin/ifconfig`)
 - o enter in the password (configured in the VNCserver above)
4. Start the VNCviewer on the remote system
 - o at the prompt, enter: **vncviewer** (on windows systems, click on the VNCviewer application)
 - o enter in the IP address of the system to be remotely controlled. Depending on the network configuration, this can be:
 - The local address of the development computer -- found by `/sbin/ifconfig`
 - The IP address of the home router/gateway (assigned by the Internet Service Provider found by going to the CGI Router Configuration Pages)
 - Another address provided by your organization's IT professional.
 - o enter in the password (configured in the VNCserver above)
5. **Important:** On the development machine, after the session has completed, shutdown the Embedix desktop tasks, and then kill the vncserver using the following command:
 - o `vncserver -kill :1`

IT Department Information:

- VNC Getting Started Guide : <http://www.uk.research.att.com/vnc/start.html>
- Technical VNC information : <http://www.uk.research.att.com/vnc/docs.html>

Summary:

The typical sequence for a Linux machine, which wants to be accessed via VNC is the following:

Machine state: The developer's machine has already booted to Linux, a graphics Xwindowing environment has been started, and the VNC server has been started for the #1 virtual screen (i.e. VNC listens on port 5801).

Action: The end users personal/corporate firewall needs to be configured to allow incoming tcp connections to the developer's computer on destination port 5801. A sample rule using iptables in linux would be similar to this:

```
iptables -A FORWARD -s 0.0.0.0/0 -d developer_ip/32 --  
destination-port 5801 -m state --state NEW -j ACCEPT
```

Note: Your corporate IT department may not approve of inbound connections from anywhere on the Net to the VNC port on your box. Please email Embedix customer support for the current Embedix source address and netmask for the above rule.

Home Network Users:

The simplified procedure below works with many home network routers and configurations. Modify as appropriate for your own hardware and network configuration.

1. Open your web browser to the router CGI interface
 - o Login in to 192.168.1.1 (substitute your router IP address for the address at the left)
2. Fill in the password
3. Go to Advanced Setup
 - o Enter into the DMZ set-up screens
 - o Enter the workstation IP address
 - o apply.
4. Remember to either disable DMZ or reset the DMZ address to an unused address when the remote connection is completed.

Replacing the Target Wizard SPLASH Screen with a new Image

Target Wizard includes the ability to replace the default Embedix Target Wizard splash screen with your own graphics image. This can be useful when providing an independent BSP to ship with your board, or when providing additional embedded components or services which are enabled for the developer.

The replacement image file should be 640x480 pixels and must be named either "splash.png" or "splash.xpm". It must be placed in: /opt/Embedix/usr/lib/images/

Debian Binary Import Tool (DBIT)

The Debian Binary Import Tool allows the developer to download hundreds of publicly available Debian Gnu/Linux binary applications, and to easily include them on target via Embedix Target Wizard. DBIT even supports dependencies, conflicts, file sizes, and help/descriptive text.

Current DBIT Features include:

- Checks the project architecture to ensure that the selected binary will in fact run.
- Debian dependencies are mapped to Embedix packages
- Extracts binary file sizes - for inclusion in storage estimates.
- Fine-grained configuration of man pages, info pages, docs and locale strings.
- Automatically places Embedix SDK package files into correct locations in the project.
- Allows user to graphically modify pre and post install scripts as well as specific dependency info.
- Installs the resultant package at the board or project level.
- Option to create LPF (Lineo Package Format) file for portability to other SDK users.

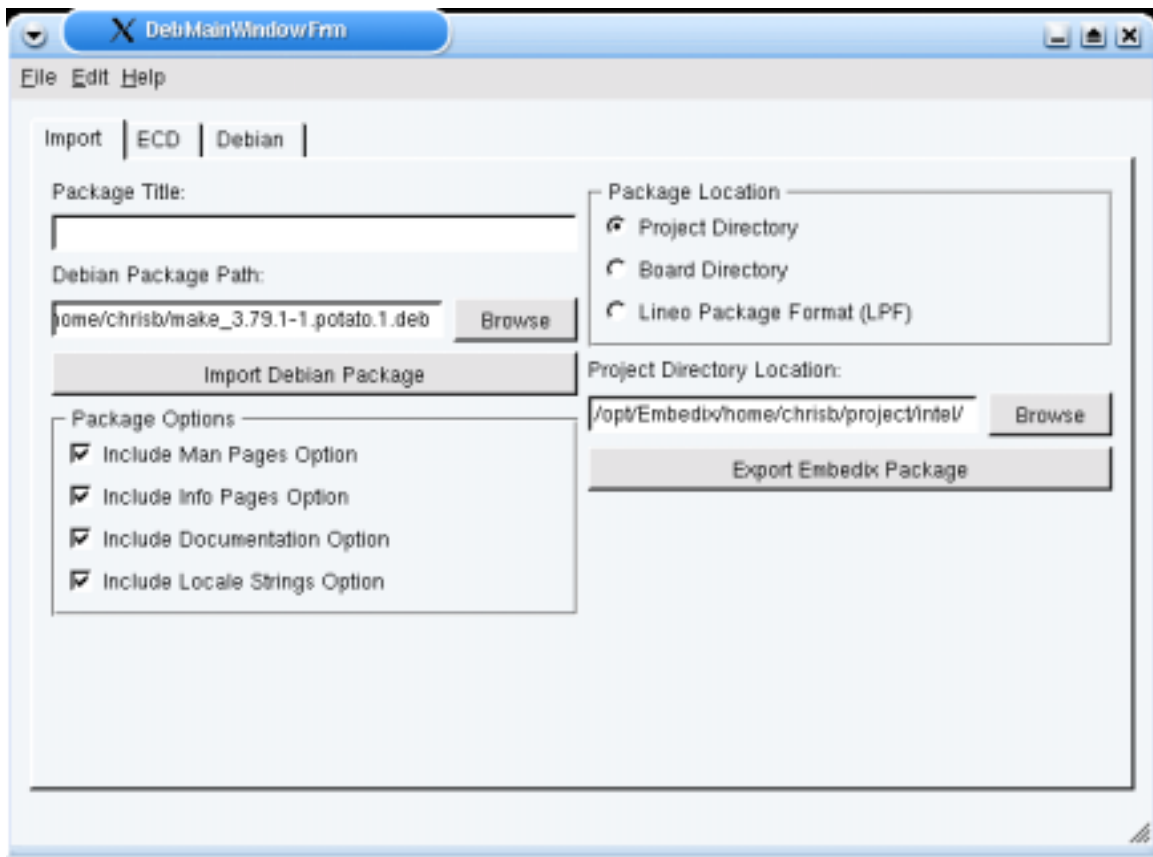
There are caveats of course. The target architecture must be compatible with the Debian binary, and the libraries must be of the same major version number.

Why Debian Gnu/Linux? Debian Gnu/Linux is a front-runner within the desktop Linux world and currently consists of 3880 different software packages ranging from network services, GUIs, word processors, multi-media utilities and much more. Among many other positive traits, most notable for the embedded world is the fact that the full Debian Gnu/Linux distribution is natively compiled across multiple architectures including x86, PPC, ARM, and more. Debian Gnu/Linux includes the *.deb package format which includes most of the information necessary for inclusion within Embedix Target Wizard. The Embedix SDK combined with Debian Gnu/Linux yields an unrivaled embedded system configuration tool, providing embedded board specific kernel and application support combined with access to a complete Linux distribution.

Importing a Debian Package

Note: Debian packages can be found at: <http://www.Debian.org/distrib/packages>

The import/export panel:



Here you can select which package to import, which options to create when importing and where to export.

To import a Debian package, enter the path in the field under the caption "Debian Package Path". You can also select the browse button to find it in your file system.

Once you've selected a package, the name will be derived from the control file in the Debian package itself, but you can override the name by entering it in the field labeled "Package Title".

When a Debian package is imported, the importer marks files as man pages, info pages, documentation, locale files, and required files. You can allow options for each of these categories in the resultant Embedix package by checking the corresponding check boxes in the group labeled "Package Options".

Note that checking these boxes does NOT mean that these files will be placed on the target at deploy time, it simply gives the user of Target Wizard the option to include or exclude them.

When you have entered the path to the Debian package and checked the Package Options boxes as desired, click the "Import Debian Package". It will take a few seconds to extract information from the Debian package.

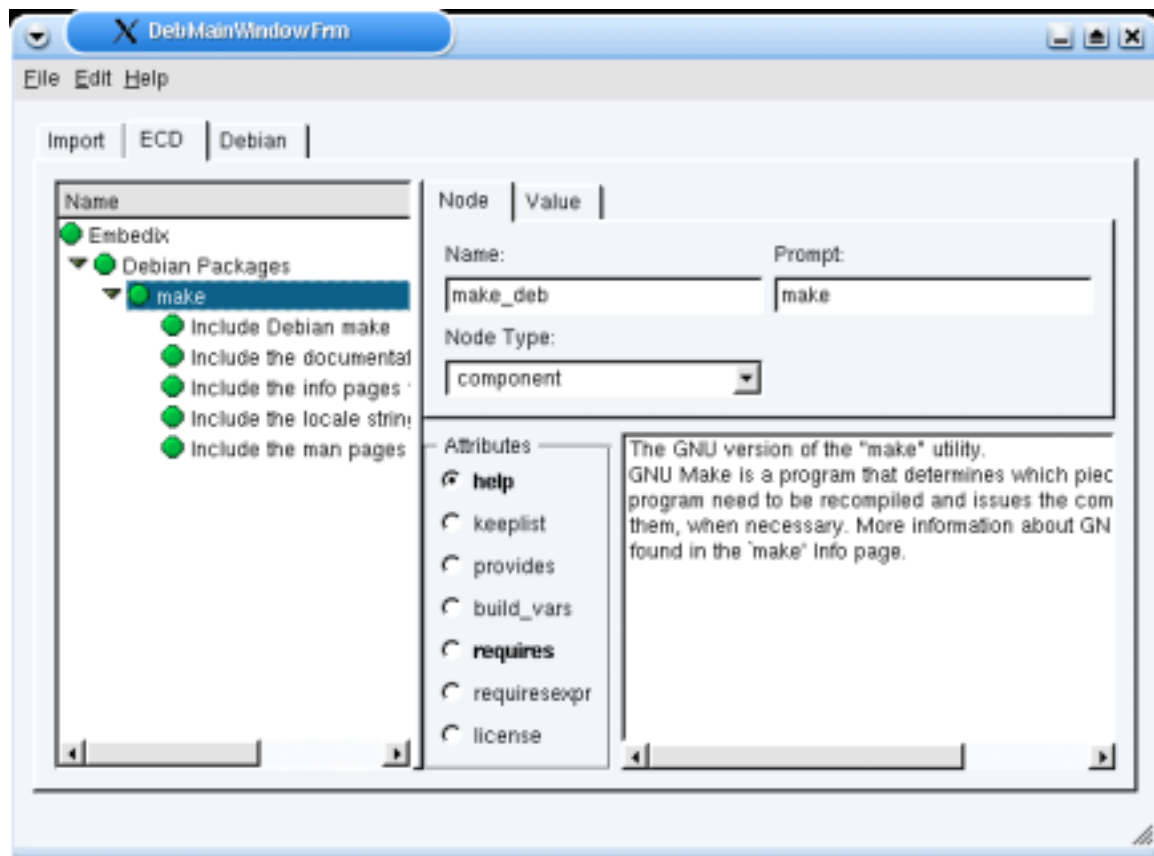
After importing the package, you can make any desired changes to the package via the ECD and Debian panels. Once all modifications have been made you then export the package to a Target

Wizard project, a Target Wizard BSP, or to an LPF which can be installed on any Embedix system, at a later time.

To export an Embedix package, select which type of export you would like by selecting the corresponding radio button in the group labeled "Package Location". Next you will need to indicate the location of the project, bsp, or where the lpf should be placed. If you selected "Project Directory" then you should enter the directory under <user home>/project/ that corresponds to your project. If you selected "Board Directory" then you should enter the directory in /opt/Embedix/bsp that corresponds to the board. If you select "Lineo Package Format (LPF)" then simply enter the directory that you would like the lpf to be placed in.

Once you have indicated the export type and location, click on the button "Export Embedix Package" and the export will take place. Instructions about the export will appear in a dialog box according to your export type.

Modifying the ECD

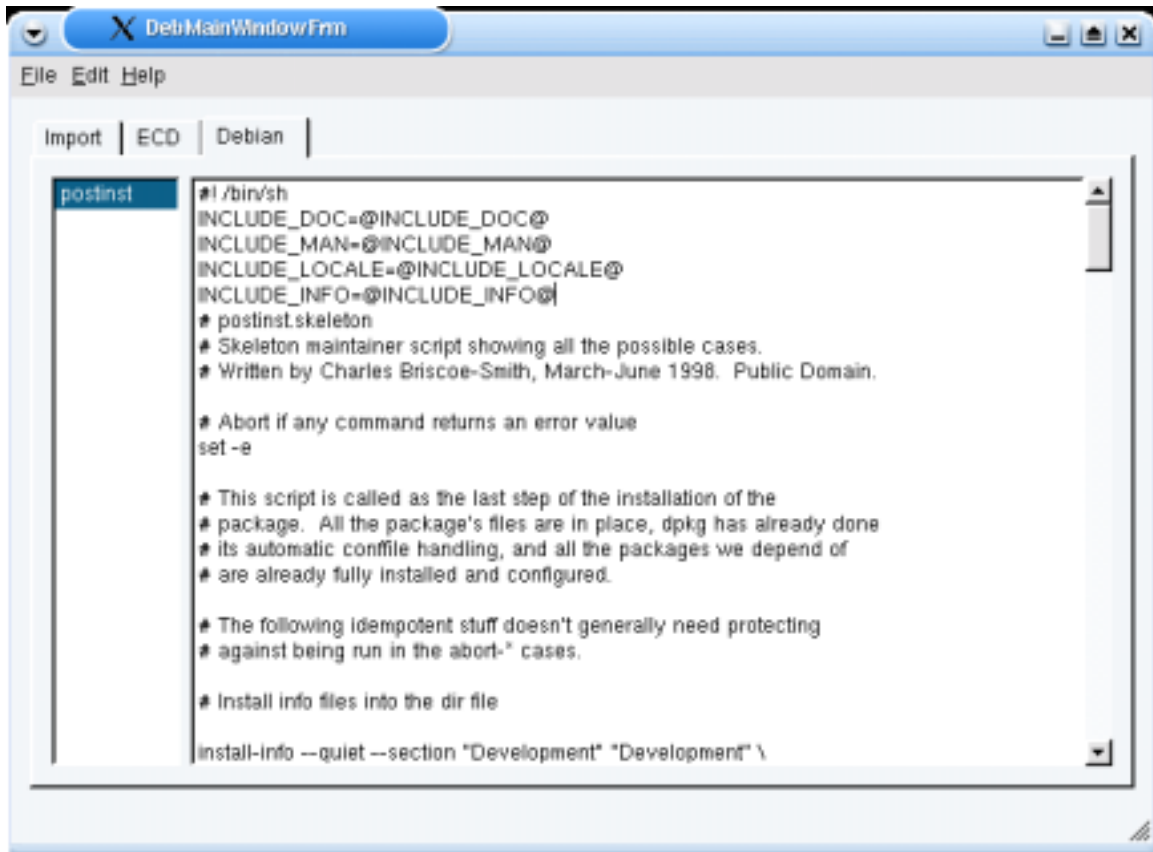


The Embedix Component Descriptor, or ECD file defines all of the elements which simplify the task of creating a project with Embedded Linux. For information about ecd files, please see the Target Wizard manuals.

Clicking on a node in the tree at the left will show its values in the fields on the right. You may change any values for nodes in the package. Most of the time no changes will need to be made in

this panel, but you may decide to add help information, change the default values of nodes, or add license information in this area. The most common changes will most likely include keeplist changes, depends information, and provides information.

Modifying the pre/post install scripts



The screenshot shows a window titled "Deb:MainWindowFrm" with a menu bar (File, Edit, Help) and a toolbar (Import, ECD, Debian). The main area displays the content of the "postinst" script. The script is a shell script for post-installation actions, including include directives, comments about its origin and purpose, and a command to install info files.

```
#!/bin/sh
INCLUDE_DOC=@INCLUDE_DOC@
INCLUDE_MAN=@INCLUDE_MAN@
INCLUDE_LOCALE=@INCLUDE_LOCALE@
INCLUDE_INFO=@INCLUDE_INFO@
# postinst.skeleton
# Skeleton maintainer script showing all the possible cases.
# Written by Charles Briscoe-Smith, March-June 1998. Public Domain.

# Abort if any command returns an error value
set -e

# This script is called as the last step of the installation of the
# package. All the package's files are in place, dpkg has already done
# its automatic conffile handling, and all the packages we depend of
# are already fully installed and configured.

# The following idempotent stuff doesn't generally need protecting
# against being run in the abort- cases.

# Install info files into the dir file
install-info --quiet --section "Development" "Development" \
```

The pre and post install scripts are used to perform actions on the target, immediately before or after installing the software. It is generally not required to modify these settings, however this panel provides an easy point at which to do this, on those occasions when it is required.

Important Note!

DBIT installs a small `deb_installer.sh` shell script in the `/sbin` directory of the target after deployment. This script **must** be run on the target, in order to enable the imported Debian applications on the target. When this script is run, it will run the individual pre and post install scripts for any Debian packages which have been imported into the system.

Once this `deb_installer.sh` script successfully runs the install scripts for a package, it saves some data so that it never runs the install scripts for that package again. This way you can run it multiple times and it does not repeatedly perform the pre-post install actions. In addition, since it keeps track of which packages have been installed, you can even add packages at a later date, run this program again, and it will run the install scripts for only the new packages, not the old ones.

Creating a New BSP

A Board Support Package for Embedix SDK consists of five primary elements:

1. Linux kernel
2. Device Drivers
3. GNU toolchain
4. User applications and Services
5. Deploy Wizard

The following paragraphs discuss how to import each of these elements into the Embedix SDK framework, in order to produce a new BSP.

Step 1: Open a SDK Project of the Same Architecture

The first step is to create a new project, from Target Wizard of a board which shares the same processor architecture as the board you intend to create a new BSP for. This is necessary so that:

1. The architecture specific GNU tool-chains are present.
2. The user applications and services are known to compile for the selected architecture.

All changes, for the new BSP will be made within this project.

Step 2: Importing a New/Patched Kernel into the Project

The 'config2ecd' is a command-line utility which maps kernel configuration logic (found in [Cc]onfig.in files in kernel sources) into an 'Embedix Component Description' file (typically named 'linux.ecd'), which in turn is used by Target Wizard or other tools to:

1. Present kernel configuration options to a user for manipulation (subject to the logic constraints of both the kernel config logic, and the .ecd constraint logic derived there from)
2. Ultimately express user choices as a .config to drive a kernel build

The linux.ecd that accompanies a given BSP works with the kernel as shipped with that BSP - config2ecd is made available so that users who wish to use a different kernel, or want to patch the "stock" BSP kernel, can create a linux.ecd that matches the configuration logic of the new/updated kernel source.

Even then, config2ecd is only needed to be used if [Cc]onfig.in logic differs between the stock BSP kernel and new/patched kernel which the user desires to configure. In such cases, regenerating linux.ecd from config2ecd is probably more straightforward than trying to modify these files by hand.

config2ecd is found in /opt/Embedix/emb-bin or on the SDK CD under miscutils/ along with a companion library, ecplib.py.

Running './config2ecd --help' will display usage information. This utility starts with a kernel source tree (default /usr/src/linux), reads a top-level arch/<arch>/config.in (default <arch>=i386), recursively descending through .../Config.in files referenced therein, manipulating the intermediate data/logic as required, to produce linux.ecd, which in turn is used by Target Wizard

to help users navigate dependencies. When the user resolves target packages and the required kernel support via `tw`, `tw` then translates the kernel-specific choices via `linux.ecd` into a `.config` file used for `'make oldconfig'`.

Note: An enhanced, graphical version of `config2ecd`, named Linux Kernel Import Tool (LKIT) is available as a complementary tool to Embedix SDK. This tool greatly simplifies the process of importing, configuring, and saving a new kernel into Embedix Target Wizard. Contact Embedix sales for information on this tool.

Example: Modifying the Existing Embedix Kernel

The `config2ecd` utility supports the re-integration of kernel patch changes back into the SDK. We'll describe this by example below.

A BSP packager, username "clem", has created a i386 project located in `~clem/project/i386`.

After making kernel source changes that involve `CONFIG_*` changes, he wishes to re-integrate this kernel back into the SDK project. To perform this, the following command should be executed at the prompt:

```
/opt/Embedix/emb-bin/config2ecd -o ./ \  
-d /home/clem/project/i386/build/rpmdir/BUILD/linux \ >& log.config2ecd
```

This command causes `config2ecd` to traverse the designated kernel source tree under the i386 project, writing the resultant `linux.ecd` file to the current working directory.

Now, to use this new `ecd` file in the existing project, or to create a new BSP, the `ECD` file should be moved to the appropriate `<project>/config-data/ecds/{local|board}/` directory of the project. (e.g. `/home/clem/project/i386/config-data/ecds/{local|board}/`)

Importing a New Kernel

It is also possible to import a completely new kernel, using `config2ecd` to create a new `linux.ecd`.

Although we have simplified the procedure as much as possible, it is important to note that this kernel replacement procedure is not a trivial one, or one which is guaranteed to be error-free with every kernel which you might download. This is because some applications or packages within an Embedix BSP have specific inter-kernel dependencies which may be broken when changing the kernel source. In many cases, the result of a new kernel import is an unfulfilled dependency icon in Target Wizard – which can be safely ignored – provided that you have enabled similar services (of perhaps a different name) within the new kernel.

Important: Prior to importing a new kernel with `config2ecd`, you need to apply all kernel patches which affect the `config.in` files. This is necessary so that these options will be available to you within Target Wizard. An example of a patch which affects the configure routines is the pre-emptible kernel patch for enhanced real-time response. This patch adds a new option (e.g. `enable pre-emptible kernel`) to the standard Linux configure scripts. However, if the patch does not affect `config.in` then it can be applied by Target Wizard when the kernel is built.

Below, we have provided an example for importing new kernels into an existing project.

Example: x86 Kernel Replacement

The steps shown below are an example of the process to update the existing Embedix x86 kernel to linux-2.4.18.

1. Download the kernel source from www.kernel.org
2. Made a directory to work with the new source in: `mkdir 2.4.18`
3. `mv linux-2.4.18.tar.gz` to the new 2.4.18 directory.
4. `cd 2.4.18`
5. `mkdir kernel` (this is used by `config2ecd` later)
6. `untar` the new source: `tar xzvf linux-2.4.18.tar.gz`
7. `cd linux`
8. Use `config2ecd` to make an `ecd` for the new kernel source: `/opt/Embedix/emb-bin/config2ecd -v -f arch/i386/config.in -d .`
9. `cd ../kernel`
10. `cp /opt/Embedix/Embedix-2.0/config-data/buildcontrol/kernel.lbc .`
11. `vi kernel.lbc`
 - - change the `"%pkg_file"` to `linux-2.4.18.tar.gz`
 - - comment out any patches specified in the `"%patches"` section
12. Open a new x86 project in TW (`x86-new_kernel-test`)
13. `cp` the new `kernel.lbc` to `<project dir>/config-data/buildcontrol/local`
14. `cp` the new `linux.ecd` to `<project dir>/config-data/ecds/local`
15. `cp linux-2.4.18.tar.gz` to `<project dir>/Packages/local`
16. `rm <project dir>/build/packagestate/kernel.*`
17. Open the new project and force rebuild the kernel
18. Load a config, build the project, deploy to a board, and boot
19. On the board: `uname -r` (should return 2.4.18)
20. Create a new BSP from the project (which includes the new kernel)
 - `/opt/Embedix/emb-bin/export_bsp --projectdir <projdir> [--bspdir <newbspdir>] [--batch] [--keeppreconfigs] [-cvsprep]`
 - e.g.:

Example: Kernel Patch, Update and Replacement

The steps shown below are an example of the process to update a stock 2.4.16 kernel from kernel.org to support the [pre-emptible kernel](http://kernel.org) maintained by Robert Love on SH architectures.

1. Download the [2.4.16 kernel source](http://kernel.org)
2. Make a directory to work with the new source in: `mkdir linux_2.4.16_sh`
3. `mv linux-2.4.16.tar.gz` to the new `linux_2.4.16_sh` directory.
4. `cd linux_2.4.16_sh`
5. `mkdir kernel` (this is used by `config2ecd` later)

-
6. untar the new source: `tar -xzvf linux-2.4.16.tar.gz`
 7. Download the patches: * [pre-emptible patch for SH](#) to the Linux directory which was created (when the kernel was untarred) under new directory created above (e.g. `linux_2.4.16_sh/linux/`)
 - o update patch for SH to the new directory created above (e.g. `linux_2.4.16_sh/`)
 8. `cd linux_2.4.16_sh/`
 9. Apply the patches
 - o `cd linux/`
 - o `patch -p1 <preempt-kernel-sh-2.4.16-1.patch`
 - o `cd ..`
 - o `patch -p1 <sh-update-rml-2.4.16-1.patch`
 - o Be sure to review the console output to verify that the patches were applied successfully.
 10. Tar up the patched kernel source
 - o `cd ..`
 - o `tar -cvf sh_linux-2.4.16.tar linux/`
 - o `gzip sh_linux-2.4.16.tar`
 11. `cd linux/`
 12. `make ARCH=sh menuconfig`
 13. Configure your kernel as appropriate for your hardware.
 - o Under the Processor type and features menu be sure to select the Preemptible kernel (NEW) option
 - o Be sure to save your kernel configuration upon exit
 14. Use `config2ecd` to make an ecd for the new kernel source:
 - o `/opt/Embedix/emb-bin/config2ecd -v -f arch/sh/config.in -d .`
 15. `cd ../kernel`
 16. `cp /opt/Embedix/bsp/BigSur-2.0/config-data/buildcontrol/kernel.lbc .`
 17. Edit `kernel.lbc`
 - o Change the `"%pkg_file"` to `sh_linux-2.4.16.tar.gz`
 - o Comment out (#) the patches files
 - o Note: You could use simply the native kernel.org tar-ball and then apply the patches here. However, because these patches add configuration items, the ECD would be incorrect and possibly non-functional.
 18. Open a new SH project in TW (BigSur-new_kernel-test)
 - o This creates the project structure into which the new kernel will be inserted.
 19. Close the new SH project
 20. `cp` the new `kernel.lbc` to `<project dir>/config-data/buildcontrol/local`
 21. `cp` the new `linux.ecd` to `<project dir>/config-data/ecds/local`
 22. `cp` `sh_linux-2.4.16.tar.gz` to `<project dir>/Packages/local`

-
23. `rm <project dir>/build/packagestate/kernel.*`
 24. Open the new project
 25. Load the configuration file produced during the `make ARCH=sh menuconfig` step.
 - Select: FILE --> Import Kernel .config
 - Browse to the `/linux` directory where the patched kernel resides (e.g. `/linux_2.4.16_sh/linux`).
 - enter `.config` in the File Name box.
 - Bug Report: This hidden file is not shown from the browse menu. Work-around: Enter in the file name manually.
 - Open
 26. Force rebuild the kernel
 27. Load a config, build the project, deploy to a board, and boot
 28. On the board: `uname -r` (should return 2.4.16)
 29. Create a new BSP from the project (which includes the new kernel)
 - See ExportBsp section below.

Step 3: Import New Device Drivers

The Package Editor tool allows one to import binaries, or build from source applications including kernel modules (i.e. device drivers). Rather than describe the process in detail, we provide a simple example below. The Package Editor documentation should be referenced for further details on use of that tool.

The example file `pe_module_example.tar.gz` on the Embedix SDK installation CD at `~/cdrom/examples/` provides each of the files shown below.

Device Driver Import Process

This section describes how to create and build a kernel module outside of the kernel source tree and then use Package Editor to include it as a selectable node of Target Wizard. There are two steps to this process. First: modify the Makefile, second: use Package Editor to create and configure the LBC and ECD files needed to integrate with Target Wizard.

Makefile changes:

- The makefile must build the program when called simply with 'make'.
- The makefile must have a clean target. You should be able to do a "make clean" and it will clean out the directory of all previously built files.
- The makefile must have an install target and it must support a prefix argument. You should be able to type "make install prefix=/`<dir name>`" and have the program install using the the specified directory as the root.
- Refer to the Embedix SDK Tools manual, page 14-15, for more specifics and an example of a makefile that will work with Package Editor.
- A text example of a `hello.c` module and makefile can be found at the end of this section. The actual `hello.c` and makefile can be found in the examples directory on the Embedix SDK CD.

Create LBC and ECD Files with Package Editor

Once your makefile is ready you can use Package Editor to create an LBC and ECD. This is a simple process consisting of a few steps as shown below.

1. Open your project in Target Wizard
2. Click on the tools button and select 'Package Editor'
3. In Package Editor, click on 'File' and select 'Import Source as a Package'.
4. In the package name box enter a name for your module
5. Browse to the directory that contains your kernel module source code and make file. and then click 'OK'.
6. Package Editor will import the source and open to the 'LBC File' page. See the Embedix SDK Tools manual, pages 26 – 37, for details on modifying the LBC for your module.
7. Click on the ECD File tab to be modify the ECD for your module. See the Embedix SDK Tools manual, pages 37 – 50 for detailed information on modifying ECDs.
8. Once you have the LBC and ECD, go to TW, click on "project ---> open recent project" and select the project that you are working in (top of the list). This will force TW to reload your project.
9. Search for your module..
10. Enable ---> build ---> deploy.
11. Test it on your target board

Kernel Module Example Source Code

```
/* start.c
 * Copyright (C) 1999 by Ori Pomerantz
 *
 * "Hello, world" - the kernel module version.
 * This file includes just the start routine
 */

/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

/* Initialize the module */
int init_module()
{
    printk("Hello, world - this is the kernel speaking\n");

    /* If we return a non zero value, it means that
     * init_module failed and the kernel module
     * can't be loaded */
    return 0;
}
```

Kernel Module Example Makefile

```
#This is the Makefile for the Package Editor kernel module example.

prefix=/
project_dir=home/coray/project/x86
kernel_dir=build/rpmdir/BUILD/linux
#source_dir=

hello.o: hello.c Makefile
    gcc -D__KERNEL__ -DMODULE -O2 -Wall -
I/opt/Embedix/$(project_dir)/$(kernel_dir)/include -c hello.c

clean:
    rm -f hello.o

install: hello.o
    install -d $(prefix)/usr/module
    install hello.o $(prefix)/usr/module
```

Step 4: Import a New GNU Tool Chain

Initially, the Embedix SDK shipped GNU tool chains which were minimally patched to facilitate cross-development and build. However, with changes to these tools and an update to our build engine, we now fully support “stock” GNU tools. This feature puts our customers in a position to replace/upgrade the Embedix provided tools at any time.

There are times when you may want to update the original Embedix GNU toolchain (i.e. gcc, gdb, ar, etc.) with newer versions. These times may include:

1. Bug fixes
2. Newer versions become available
3. New optimizations are required to support your new BSP.
4. Your own engineers have integrated custom changes as required by your project.
5. Given libraries/kernels may require specific tool versions.

The GNU tool chain consists of a number of different utilities which facilitate application build, link, assembly, and debug (among many other tasks).

Note: The Embedix SDK CD includes documentation on these tools in the following directories:

`/documents/gcc.gnu.org/onlinedocs/...`

`/documents/www.gnu.org/binutils-2.12/html_node`

Typically (according to architecture type and language support), these GNU tools include the following:

GNU Tool Elements

The lists below provide information for what elements are generally included within the “gnu tool element” category. Those designated with a “*” are mandatory for functioning within Embedix SDK for compilation of C applications and the kernel. Other components can be considered to be optional (though highly recommended).

GCC:

Gnu Compiler Collection (See : <http://gcc.gnu.org/>)

GCC consists of a number of different applications or programs which collectively support a diverse set of compilation scenarios. Common files appearing within Linux systems include:

- c++
- cc *
- cpp *
- g++
- gasp
- gcc *
- gcov - gcov is a tool you can use in conjunction with GNU CC to test code coverage in your programs.

GDB:

Gnu Debugger (See: http://www.gnu.org/manual/gdb-5.1.1/html_node/gdb_toc.html)

GDB enables dynamic or post-mortem debugging of programs. A graphical “front end” for GDB called the Data Display Debugger [DDD] can be used in conjunction with GDB if desired.

Binutils:

The GNU Binutils (see: <http://www.gnu.org/software/binutils/binutils.html>) are a collection of binary tools. The main ones are:

- ld * - the GNU linker.
- as * - the GNU assembler.

But they also include:

- addr2line * - Converts addresses into filenames and line numbers.
- ar * - A utility for creating, modifying and extracting from archives.
- c++filt - Filter to demangle encoded C++ symbols.
- gprof - Displays profiling information.
- nlmconv - Converts object code into an NLM.
- nm * - Lists symbols from object files.
- objcopy * - Copys and translates object files.
- objdump * - Displays information from object files.
- ranlib * - Generates an index to the contents of an archive.
- readelf * - Displays information from any ELF format object file.
- size * - Lists the section sizes of an object or archive file.
- strings * - Lists printable strings from files.
- strip * - Discards symbols.
- windres - A compiler for Windows resource files.

Other Software:

Other software necessary for the functioning of a cross-compile environment within Embedix SDK include:

- depmod * -
- ldconfig * - configure the a.out shared library cache
- ldd * - list dynamic object dependencies
- protoize
- unprotoize

Locating Updated GNU Tools

Since GNU tool chains are built from open sources, they can be obtained from various internet locations or from alternative embedded Linux providers.

As of the writing of this document, some good places to look for these tools include:

Architecture	URL
ARM (including xScale)	http://www.emdebian.org/downloads.htm http://www.ocdemon.com/OCDemonLinux.html http://www.arm.linux.org.uk/developer/tools.shtml http://www.lart.tudelft.nl/lartware/compile-tools/
PPC	http://www.emdebian.org/downloads.htm http://www.ocdemon.com/OCDemonLinux.html
MIPS	http://www.ocdemon.com/OCDemonLinux.html
Toshiba TX49	http://www.ocdemon.com/OCDemonLinux.html

Gnu Tool Naming

A specific tool chain is tied to specific versions of BSP(s). That is, a BSP for a MIPS architecture requires that MIPS cross-compile tools be used.

Embedix SDK uses architecture specific cross-compiler prefixes to specify the appropriate versions of cross-compile tool chains.

Tools prefixes currently employed by Embedix BSPs include the following:

Tools Prefix	Architecture
i386-linux-	x86
arm-linux-	ARM
m68k-elf-	68k
mipsel-linux-	Little Endian MIPS
native-linux-	x86 host
powerpc-linux-	Power PC
sh3-linux-	Hitachi SH3
sh3eb-linux-	Hitachi SH3 Big Endian
sh4-linux-	Hitachi SH4

Tools Prefix	Architecture
sh4eb-linux-	Hitachi SH4 Big Endian

New tools intended to replace the default Embedix tools should be named with the same prefix and placed in the appropriate directory as shown below. Tools intended to be used for only a specific set of BSPs can be named with an arbitrary prefix, provided that the BSP configuration file (described below) includes that arbitrary prefix in its “%cross” field.

The cross-compiler prefix used by any one Embedix BSP can be found by looking at the bsp_config file located in /opt/Embedix/bsp/BSP_NAME/config-data/buildcontrol/bsp_config and noting the value of the “%cross” field. Alternatively, you can view this from the BSP Configuration Wizard, called from target wizard by:

FILE -> RUN_WIZARD -> BSP CONFIG EDITOR

New Tools Integration Steps

When installing different GNU toolchains, you should confirm that all elements from the “GCC”, “GDB”, “binutils” and “Other Software” categories above are present with the architecture specific prefix when installing them to the directories noted below.

Important Notes:

In general, Tools listed within the GCC and binutils categories may NOT be “mix and match” combined. That is, the updated tools software should include the full suite of applications required. It should not rely on a new piece here and an older piece there. These must be internally consistent as a set.

Those elements listed in the “GDB” and “Other Software” categories may often be mixed with newer GCC and binutils versions.

GCC Elements:

The Embedix SDK builder utilizes search paths using software from the first location in a path list where those tools are found.

Embedix default tools shipped with individual Embedix Board Support Packages (BSPs) are placed in: /opt/Embedix/tools/bin. Replacement toolchains may be installed anywhere under /opt/Embedix, as long as appropriate “%cross” and “%xdir” directives are placed in a governing bsp_config or .lbc file (depending on project vs. package scope for use of the tools).

Note: Since the Builder.pm machinery operates while chrooted to /opt/Embedix, using “/opt/Embedix” as a path prefix in the “%xdir” path is transparent.

Builder.pm may be found in multiple locations:

```
/opt/Embedix/home/<user>/<project>/emb-bin/Builder.pm
```

```
/opt/Embedix/bsp/<board>/emb-bin/Builder.pm
```

```
/opt/Embedix/emb-bin/Builder.pm
```

If modification of `Builder.pm` is called for, you should modify only one of these files. You should modify the `<project>` version if present. If there is no `<project>` version then you should modify the `<board>` version. If no `<project>` or `<board>` version is present then you can modify the `/opt/Embedix/emb-bin/` version, or preferably copy it to `<project>` or `<board>` and modify the copy.

Note: Although you have installed a new tools version, the old tools are still in place and you can usually explicitly use them (on a per package basis) by modifying the package's LBC with something like:

```
CC=/opt/embedix/tools/bin/mipsel-linux-gcc
, or by using "%xdir" / "%cross" directives in <pkg>.lbc
```

GCC from other Linux Sources

Often it is possible to download a working architecture tool chain from other Linux providers, open source web repositories, or from silicon manufacturers.

These tool chains, provided stand-alone, or incorporated into a larger "BSP", are often delivered in the form of "tarballs" or RPM files.

Tools provided as tarballs can be placed within Embedix SDK by doing something like the following:

```
cd /opt/Embedix
tar -tzvf <path>/random_toolchain.tar.gz
    to check the sanity of paths
tar -xzvf <path>/random_toolchain.tar.gz
Edit "%cross" / "%xdir", in the bsp_config or <pkg>.lbc file to match the new
toolchain target prefix.
```

Tools provided in RPM format can be used if they have been packaged as re-locatable RPMs. If these are re-locatable, then it is a simple matter to install these tools to the proper directory by a command such as:

```
rpm -ivh --prefix /opt/Embedix/<desired_location>
your_tools.i386.rpm
```

Cross Development Tools provided by other Linux vendors which do not install to `/opt/Embedix/usr/local/bin` can be used if an `"%xdir"` field is added to an appropriate `bsp_config` (for project scope) or `<pkg>.lbc`.

```
Example:      %xdir
              /opt/Embedix/usr/local/mipsel-linux/bin
```

TimeSys (TM) tool-chain import example:

TimeSys (<http://www.timesys.com>) is an embedded Linux provider which supplies board support packages for free download. These BSPs generally include a number of hooks for TimeSys'

development and analysis tools and for their real-time solutions. The full GNU tools suite is provided as a part of these BSPs. A typical TimeSys BSP is packaged in two separate tarballs.

The steps shown below are typical for importing a TimeSys provided tool chain into the Embedix SDK.

For example, the TimeSys 3.1 MIPS BSPs has `tslinux-3-1-vr5500-bsp239.tar.bz2`.

1. Download `tslinux-3-1-vr5500-bsp239.tar.bz2`.
 2. `tar xjvf tslinux-3-1-vr5500-bsp239.tar.bz2 -C /tmp`
 - 2.5. `tar tjvf /tmp/tslinux-3-1-vr5432-bsp239/crosstools/toolsuite.vr5432.tar.bz2`
- look for naming collisions with anything already in `/opt/Embedix/`

If safe (or after saving off colliding files):

3. `tar xjvf /tmp/tslinux-3-1-vr5432-bsp239/crosstools/toolsuite.vr5432.tar.bz2 -C /opt/Embedix`
4. add `"%xdir" / "%cross"` directives in build control files (see discussion above)

MontaVista Software (TM) tool-chain import example:

MontaVista Software (<http://www.mvista.com>) is an embedded Linux provider producing "BSPs" which generally include enhanced real-time performance. A typical MontaVista BSP consists of a set of RPM files which install, among other items, the cross-compile tool chain to the host development platform.

For example, a version of a MV BSP for PPC 7xx has a `redhat62/hhl-cross-ppc_7xx-gcc-2.95.3-hhl2.0.2.i386.rpm`

1. `rpm -qpl hhl-cross-ppc_7xx-gcc-2.95.3-hhl2.0.2.i386.rpm`
2. Observe that rpm contents will be installed to `/opt/hardhat`
3. `cd /opt; ln -s Embedix hardhat; cd /opt/Embedix; ln -s . hardhat`
4. `rpm -i --nodeps hhl-cross-ppc_7xx-gcc-2.95.3-hhl2.0.2.i386.rpm`
5. add `"%xdir" / "%cross"` directives in build control files (see discussion above)

Building your own Tool Chain

Building tool-chains is a non-trivial process and should not be attempted by the novice developer. The GCC documentation located on the Embedix CD provides adequate information on this process, however a few additional hints specific to integration with Embedix SDK are provided below. Note that this is not meant to be a comprehensive tutorial on building tool chains:

- When configuring the toolchain, use
`"--prefix=/opt/Embedix/<your_choice>"`
- When designating `libc` and kernel headers to be used in the toolchain, designate headers from your BSP's kernel and `libc` sources.

Binutils:

A cross-assembler and cross-linker as provided by binutils are required by the Embedix cross-build engine. If alternate binutils are needed, they can be placed anywhere under /opt/Embedix (with appropriate "%xdir" / "%cross" directives in build control files if they don't wind up in usr/local/bin/).

/opt/Embedix/usr/local/bin

Below is a table of the tools you should minimally put in this directory:

- `as' This should be the cross-assembler.
- `ld' This should be the cross-linker.
- `ar' This should be the cross-archiver: a program which can manipulate archive files (linker libraries) in the target machine's format.
- `ranlib' This should be a program to construct a symbol table in an archive file.

Other binutils files should be placed in this same directory.

Libraries:

If you want to install libraries to use with the cross-compiler on Embedix SDK, such as a standard C library, put them in the directory /opt/Embedix/<your_choice>/lib

/usr/local/target/lib'; installation of GNU CC copies all the files in that subdirectory into the proper place for GNU CC to find them and link with them.

Header Files

Architecture specific header files should be placed under the directory /opt/Embedix/<your_choice>/include, before building the cross compiler.

Modifying tc.config

The directory /opt/Embedix/tools/ contains a number of files ARCHNAME.tcconfig. This file was used to specify parameters for old style [Lineo modified] toolchains. This file is not necessary for imported toolchains since "%toolchains" is superceded by "%cross" in the bsp_config file.

Testing New Tool Chains

It is advisable to perform a test of new tool chains prior to a build within Embedix SDK.

As a simple test, a hello world application can be written, and the toolchain exercised, using the Package Editor. Please see Embedix SDK Tools user guide for further instructions on Package Editor usage.

Step 5: Import New Applications and Services

One of the advantages of starting from an existing Embedix BSP is that the you already have a proven set of applications and services which run the target architecture.

If you desire to integrate more functionality into your BSP, you should reference the Package Editor or DBIT to import the following application types:

- Device Drivers
- Build-from source applications
- Binary applications

You can also easily import new applications or configuration files by simply copying them to the project's /merge directory. Any file which appears underneath of the /merge directory will be placed into the same location on the target root file system. If duplicate file names occur, those under the /merge directory supercede (i.e. over-write) those which are provided or generated via Target Wizard.

Step 6: Modify the Deployment Wizard

The final component of a BSP for Embedix SDK is the deployment wizard. While not required (i.e. you can always perform a manual deploy), this feature allows complete simplification of the development process, allowing the customer to focus on their own applications rather than the details of board support. For those who wish to complete deploy to the target device without having to write this deployment wizard, we've included a section describing how to do a manual deploy at the end of this section.

Deployment Architecture

This section explains the architecture of how deployment scripts are used and called by Target Wizard.

Important Files

- **/opt/embedix/emb-bin/emb_deploy** - This is the python script that is called directly by TW. It determines what script to call as the deployment script. It first looks in the <project>/emb-bin directory to see if any deployment scripts exist there (any scripts that end in _emb_deploy) and invokes that script if it exists. Otherwise it invokes a generic deployment wizard that offers minimal functionality.
- **/opt/embedix/emb-bin/generic_emb_deploy** - this is a generic deployment wizard that offers the ability to generate the file system for that target as a directory tree in a location of the user's choice.
- **/opt/embedix/bsp/<bsp-name>/emb-bin** - This is where board specific deployment wizards should be placed. All files in this directory will be copied to <project>/emb-bin when a project is created for this board
- **<project>/emb-bin** - This is where the deployment script for a specific project resides.
- **/opt/embedix/emb-bin/pagelib.py** - This is a library of commonly used pages. This includes pages to build the file system and show the log, run lipo, etc. To use these

pages, import pagelib and subclass specific pages, overriding methods and variables where necessary. Detailed instructions are given below.

Sequence of events when deploying

When the user wants to deploy to the target for testing, he can invoke the deployment script in one of two ways.

1. Click on the deployment button in Target Wizard.
2. run `/opt/embedix/emb-bin/emb_deploy` from the command line with project and board arguments:
 - o `/opt/Embedix/emb-bin/emb_deploy --projectdir /home/chrisb/project/intel --board i386_default`

Either method will start the script `/opt/embedix/emb-bin/emb_deploy`. This script will then scan the `<project>/emb-bin` directory looking for deployment scripts (anything ending in `_emb_deploy`). If it finds one then it will run that script and exit. If it doesn't then it will run the generic script located at `/opt/embedix/emb-bin/generic_emb_deploy`. It's important that only one deployment script be located in `<project>/emb-bin` because if two exist, `emb_deploy` will run the first one it finds and that might not be the one that you want. Also note that the deployment script in `<project>/emb-bin` should not have a `.py` suffix.

Deployment Phases

There four main "phases" that should occur when a deploying. Here is a general description of each and possible steps that would occur in each phase.

- **Root FS Prep** - This includes things that prepare the image (file system, kernel, etc) for deployment. It should only include steps that are orthogonal to the method of deployment. (ie don't do something here that would only need to be done when doing an NFS deploy) Possible steps for this phase include:
 - o Build the root file system
 - o Run lipo on the resulting file system
- **Host Setup** - This includes things that prepare the host system for deployment. Depending on what deployment methods are being used in the script, this step might not even be needed (if you are doing deployment on floppy disks for instance). Very often, these steps will only need to be performed once on the host machine. Possible steps for this phase include:
 - o Gather host information (ip address, hostname, etc.)
 - o Start host services (tftp, NFS)
 - o Modify configuration files on host.
- **Target Setup** - This includes gathering information about the target system that will be needed prior to deployment. As in the Host Setup phase, this may be optional depending on deployment type. Possible steps here include:
 - o Gather target information (MAC address, target address, etc.)
 - o Giving instructions to the user about how to configure firmware on the target
- **Deployment Action** - This phase is where the deployment actions actually take place. Possible steps here include:
 - o Making a deployment boot disk for target
 - o Flashing the kernel and or file system to target memory

-
- Giving instructions to the user about how to tftp the kernel to the target and boot it.

A note about host setup:

Often, part of deploying will include modifying host configuration files and starting or restarting host services. One example of this is NFS deployment. To deploy using NFS, the `/etc/exports` file needs to be modified and NFS needs to be restarted. You should never modify host configuration files without asking the user first. The best way to do this is by making it an option with a checkbox. When making modifications to a configuration file, you should add a comment where you've made the modification and should explain what changed and why. In addition, you should make a backup version of the configuration file so that the user may revert to the old configuration if desired. Use a numbered backup so that the next time the deployment is run it doesn't backup over the original backup file. Routines are provided by the wizard system and `pagelib.py` to make it easier to perform these steps.

Writing a deployment wizard

Getting Started

In order to write a deployment wizard, you need to know python and understand the wizard framework. Python is a scripting language similar in power to perl, but with a syntax that is a bit more readable and maintainable. It also is strongly object oriented. If you are familiar with programming, you should be able to learn python in about a day with a good book.

The deployment wizards also make use of QT bindings for Python. A good starting point is to reference the PyQt documentation located at:
<http://www.riverbankcomputing.co.uk/pyqt/index.php>

In addition, there are a series of example wizards that get installed with the wizard system at `/opt/embedix/lwiz1.0/examples`. These wizards are fully commented and range from the most basic wizard possible to a somewhat complex wizard with interdependencies, progress bars, and many types of widgets.

Once you understand the basics of python and the wizard framework, the best way to write a deployment wizard is to read and understand the rest of this documentation and then look at some real deployment wizards to see how they use all of these concepts. Most likely the best files to look at are:

- `/opt/embedix/emb-bin/generic_emb_deploy`
- `/opt/embedix/emb-bin/pagelib.py`

You can also take a look at board specific deployment wizards. These are located in:

`/opt/embedix/bsp/<board-name>/emb-bin/*_emb_deploy`

Using the page library (pagelib.py)

There are many tasks that are common to deploying, independent of board type and deployment method. These include:

- Building the target file system as a directory tree
- Running `lipo`

In order to reduce duplication of effort, a library of commonly used pages has been created in `/opt/embedix/emb-bin/pagelib.py`. This library consists of pages that may be sub-classed in a

deployment wizard. Some pages require that you place specific values in variables or the database for the pages to work. One such page is the lipo prompt page. Here is an example of how to use that page in a wizard of your own design.

If you look at the source for these pages in pagelib.py you'll read the following documentation:

```
__doc__ = """
    to inherit from this page, the database must have:
    TARGETFS - root of the directory tree
    PROJPATH - path to the project
    ARCH - the architecture of the project (right now mipsel-linux
and i386-linux)
    """
```

That means that we'll need to set those values in the database in order to use this page. Here is the code that uses both lipo pages from the i386 deployment wizard:

```
import pagelib
class lipo_prompt_page(pagelib.lipo_prompt_page):
    name = "lipo_prompt"

    def post(self, database):
        database["TARGETFS"] = database["PROJPATH"] +
"/tmp/rootfsdir"
        database["ARCH"] = "i386-linux"
        pagelib.lipo_prompt_page.post(self, database)
        return 1

class lipo_lib_page(pagelib.lipo_lib_page):
    pass
```

The first thing we do is subclass `pagelib.lipo_prompt_page`, thus inheriting all methods and variables from that class. Every page includes its own name, step, description, and `help_string` variables, but we decided to override the name variable. We also overrode the post routine to set the needed values in the database (`PROJPATH` was a value set in an earlier page so we don't need to set it here) The post routine then calls the post routine of the super class (`pagelib.lipo_prompt_page`). It's important that if you override a method, you still make a call to the original page's method or it might not do what you expect. In this case lipo is run in the post method so we need to call it since it is not called if we override it.

Notice that we didn't make any changes to the `lipo_lib_page` at all. The `pass` is just there because a class must contain at least one statement and `pass` acts like a no-op. It's important that you sub-class a page even if you don't need to make any changes to it. That way, if you need to make changes, you can simply add the modifications later to the subclass. Also, if you don't subclass it, that page is not brought into the local scope of the wizard and it won't be instantiated by the wizard engine.

It's important that `pagelib` be imported by the statement: `import pagelib` and not `from pagelib import *`

In the latter case, all pages in `pagelib` would enter the namespace of the wizard module and they would all be instantiated by the wizard engine which would slow things down. Even worse, there is the possibility that you might have a page class with the same name as a class in `pagelib`, or there may be two classes that have their name variable set to the same string. This would wreak havoc when the wizard engine tried to display a page with that name.

Running shell commands from within the wizard

Most often when trying to deploy an image, you will need to execute a number of shell commands. The best way to accomplish this in python is through the use of the `os.system` call. This method will execute a script in a sub-shell and return the exit code of the command. The string need not be limited to one line in length. here is a brief example:

```
#we're assuming here that database already has entries
#for PROJPATH, BOOTDIR, KERNELPATH, and INITRDPATH
import os
command = """
cd %(PROJPATH)s
if [ ! -d %(PROJPATH)s/%(BOOTDIR)s ] ;
then
    mkdir -p %(PROJPATH)s/%(BOOTDIR)s
    if [ 0 -ne $? ] ; then exit 1 ; fi
fi
cp %(KERNELPATH)s %(PROJPATH)s/%(BOOTDIR)s
if [ 0 -ne $? ] ; then exit 2 ; fi
cp %(INITRDPATH)s %(PROJPATH)s/%(BOOTDIR)s
if [ 0 -ne $? ] ; then exit 3 ; fi
""" % database

status = os.system(command) >> 8

if status:
    print "There was an error on command #", status
```

The command string is created using triple quotes, meaning that you can use newlines and single quotes without problems. it also uses string substitution from the database. The `%(PROJPATH)s` in the triple quoted string will be replaced with value of `database["PROJPATH"]` provided that database is a python dictionary and the entry "PROJPATH" in that dictionary is a string. Make sure that you add `% database` after the string and that all entries you use in the command string are also in the database or you'll have problems.

Also, you'll notice that the shell fragment checks the status code of each command and exits with a specific code if it is non-zero. The line:

```
status = os.system(command) >> 8
```

runs the command and returns the exit status code (this will be zero if everything went fine). Don't worry about the `>> 8`. Just know that you need it because of the way that `os.system` works.

Running as Root

Deploy scripts should NOT use `suwrapper`. Instead, you should include the following magic string somewhere in your script (preferable at the top, below `#!/opt/Embedix/lwiz1.0/usr/bin/python` line.) and the whole script will be run as root (thus eliminating the need for `suwrapper`).

```
***RUN AS ROOT**
```

This string must be on it's own line and must begin at the first character of that line. That is, this will work:

```
***RUN AS ROOT**
```

... and this will not

```
***RUN AS ROOT**
```

Because including this string will cause the script to be run as root, the script should not check the uid of the process. Many old scripts would check to see if they were being run as root and display a warning box if they were. If you have copied your code from an old deployment script, you need to remove this check.

Utility Functions

In addition to pagelib.py there is also a python module, wizutil.py, that contains methods to accomplish many common tasks, these include:

- parsing and writing configuration files - these are used to write the a database to a file so that values entered in once by the user can be persistent.
- string substitution in files - used to alter configuration files
- making numbered backups of files
- checking to see if a string is in a valid ip address format
- showing a busy indicator window when executing a long command

Look at the file located in /opt/embedix/lwiz1.0/usr/lib/python2.0/site-packages/wizutil.py to see each of the utility functions. Each function contains comments about usage.

Deployment Guidelines

You can use the following as a checklist of features (or feature attributes) in your deployment script:

- Don't have a first page that consists of just "Welcome"
- Do follow the general sequence:
 - Root FS Prep, Host Setup, Target Setup, Deploy Action
- Don't modify host configuration files without asking the user
- Do provide a checkbox and allow the user to:
 - skip modification of host configuration files
 - see detailed instructions for modifying host configuration files
 - (see nfs pages in pagelib.py for an example)
- Do make incremental backups of modified host configuration files
- Do annotate modifications to host configuration files
- Do provide at least the basic "install to directory" option in EVERY deployment wizard
- Do utilize as much functionality as possible from pagelib.py, in order to have consistent look, feel, and operation of your script.
- Do put board-specific auxiliary programs in <board>/emb-bin
- Do put the script itself in <board>/emb-bin, with a name ending in "_emb_deploy".
- Do include the following magic string somewhere in the deployment script on it's own line (preferable on the top) so that it is run as root (regardless of who started Target Wizard):
 - ```
***RUN AS ROOT**
```

- 
- Do NOT use the "running as root" warning dialog at the beginning of the script. The scripts now run as root by default.
  - Do NOT use suwrapper for individual sub-commands run by the script (this is not necessary, since the script now runs as root)

## Deployment Hooks

### ***Requirements***

Allow hooks into deployment so that deploys can be modified without changing the python source

### ***deploy\_page***

The standard page class used in the wizard framework has been sub-classed. The new page class is called `deploy_page`. This new class allows hooks for pages and packages. The `post` method of the page class has been overridden and does the following:

- call the `validate` method (will return 1 if OK and 0 if not OK)
- call `save_config` method
- call the `do_action` method with `pagename + _action` (this is where the action happens)

The two new methods, `validate` and `action` have both been defined. Users who wish to do their own validation need to override the `validate` method. The `action` method should typically not be overridden as it is the method that looks for and calls the page and package specific hooks.

You may override the `post` method and make your own calls if you like. It should be easy to call `do_action`, `validate`, and/or `save_config` from anywhere in the `post` method if you want. In addition, if you would like to have the above `post` functionality, but want to do some processing or logic beforehand then you can call the super-class method:

---

example:

```
class foo_page(deploy_page):
 #normal stuff here
 def post(self, database):
 #do some logic, processing, etc.
 #now call the super-classes post method
 return deploy_page.do_action(self, database)
```

below is a brief rundown on the added methods:

### **Validate(self,database)**

Validate by default just returns a 1 indicating that everything is ok. If a user wishes to validate entries received from the database, they may be examined here. Validation Boxes may be used as well. If you override this method, make sure that you return a 1 if the wizard may proceed and a 0 if you wish to stay on this page. If a 1 is returned then the database will be saved to a config file (for persistence if the wizard is cancelled) and the action method is called. Here is a brief example:

```
class foo_page(deploy_page):
 #normal stuff here for layout, etc

 def validate(self, database):
 if not os.path.exists(database["KERNEL_PATH"]):
 ValidationBox("Bad Kernel Path", ""
 You did not enter a valid kernel path.
 Please enter a location that really does exist!""")
 return 0
 return 1
```

### **save\_config(self, database)**

This method saves the all of the database values to a deployment config file:

```
<project>/build/projectstate/deploy.conf
```

This file is automatically loaded by the `deploy_setup()` function that should be called in the `start()` routine of a wizard that uses this framework.

### **do\_action(self, database, action)**

The action method is the basic hook mechanism. It scans the `deploy.ldc` file for a section that matches the parameter `action` that was passed in. If it finds that section then it executes the section in the log window with the values in the database exported to the environment.

This means that if `database["FOO"] = "BAR"` in the python script, you can use `$FOO` in the shell fragment for that section.

The default post method makes the call:

```
do_action(self, database, self.name.lower() + "_action")
```

So if a page has the name `build_fs`, then post runs the action `build_fs_action`.

In addition, if the section of the `deploy.ldc` file that is to be executed contains the magic string:

```
RUN FROM PACKAGE LBC %<section>
```

Then it will make the call:

---

```
self.do_package_sections(database, <section>)
```

### **do\_package\_sections(self, database, lbc\_section)**

When this method is called, it will make a list of all packages that are turned on for the project. For each package, it will look for an lbc (in local, board, generic order) that has the specified section in it. If that section is found then it is run with the database values exported and the buildvars for that package exported to the environment as well.

### **Usage notes**

The "hooks" feature allows people to modify deployment behavior without changing the python deployment scripts. This makes it possible for people to add functionality without knowing python or anything about the wizard architecture.

### **Importing deploy\_page**

Because this framework is intended to be used by deployments for BSPs and is not intrinsically part of the wizard system, you will need to import `deploy_page` yourself in the python deployment script. If you are using the Embedix SDK 2.4-1 then `deploy_page` will be in a standard location that python will find. The code to import `deploy_page` is below:

```
Stuff needed to use deploy_page classes and functions
#append the project's emb-bin directory to the system path
import sys, os
if len(sys.argv) < 2:
 print "The project directory must be specified on the command line"
 sys.exit(1)

try:
 from deploy_page import *
except:
 print "Could not import deploy_page.py"
 print "project directory is", sys.argv[1]
 print "sys.path = ", sys.path
 sys.exit(1)
done with deploy page stuff
```

You should place this code at the top of your wizard script in order to import the `deploy_page` correctly. If `deploy_page.py` is in your `<project>/emb-bin` directory and the `deploy` script is being called with either `emb_deploy` or from Target Wizard then this will definitely work.

### **Calling deploy\_setup(database)**

In order to set up the `deploy` framework, the function `deploy_setup` must be called with the `database` as the argument. This function takes care of the following tasks:

- parses `sys.argv` and sets `database["PROJPATH"]` and `database["EMB_BIN"]`
- loads `deploy.conf` file (this file holds old database values for persistence)
- checks for existence of lock file and shows a dialog if it does exist
- creates lock file and sets the exit function to remove lock file
- removes any old log files if they exist

- sets database["logfile"] to be <project>/deploy.log
- adds /opt/embedix/bin and /opt/embedix/emb-bin to front of \$PATH

You should call this from the start method of your wizard. If you plan on doing nothing else then the code should look like this:

```
def start(database):
 return deploy_page.deploy_setup(database)
```

## The LDC File

When using the deployment hooks, the main file of concern is:

```
<project>/build/projectstate/deploy.ldc
```

This is an lbc-style file that contains actions to occur after each page is executed. When the **next** button is clicked on a deploy page, the wizard calls the validation routine for that page (if there is one) and then looks at the deploy.ldc file for a section called <page\_name>\_action. If that section is found then it is executed in a log window. Here is a brief example from a deploy script:

```
---- extracted from the deploy python script ----
class build_fs_page(deploy_page):
 name = "build_fs"
 step = "Build Filesystem"
 description = "This step builds the filesystem that will be
placed on the target."

 def layout(self, database):
 self.add_widget(label_widget("""
When you click on Next, the filesystem will be built. This may take a
minute or so. The log of the build will be displayed in a separate
window. When the build is done, the root of the target directory tree
will be %(PROJPATH)s/tmp/rootfsdir. You may examine the log of the
build. Afterwards click OK on the log window proceed"" % database))
 self.add_widget(checkbox_widget("Rebuild Filesystem",
"REBUILD_FS", "1"))
 self.add_widget(checkbox_widget("Add Merge Directory
Contents", "ADD_MERGE", "1"))
 self.add_widget(checkbox_widget("Add Kernel to boot
directory of target filesystem", "ADD_KERNEL", "1"))

---- below is a section of the deploy.ldc file ----
%build_fs_action

if ["${REBUILD_FS}" = "0"] ; then option1="--nobuildfs"; fi
if ["${ADD_MERGE}" = "0"] ; then option2="--nomerge"; fi

/opt/Embedix/emb-bin/emb_mktimage.py --projectdir ${PROJPATH}
${option1} ${option2}
if ["${ADD_KERNEL}" = "1"] ;
then
 mkdir -p ${PROJPATH}/tmp/rootfsdir/boot
 cp ${IMAGEPATH} ${PROJPATH}/tmp/rootfsdir/boot
fi
```

---

The action name is `build_fs_action` because the page name is `build_fs`. It is important that page name's do not contain spaces in them. Since the user only sees the page step string in the title of a page and not the name, it should not be difficult to keep spaces out of the name.

Notice In the above example, `IMAGEPATH` has already been set and `PROJPATH` is always in the database by default as the project directory path. Notice that the variables used for the widgets in the python script can be used as shell variables in the section of the `ldc` file. You can reference any database value as a shell variable in these sections. They will always be exported as strings so `5` will become `'5'`. If you wish to call a seperate binary file, you can call it from within this section. `$EMB_BIN` is a shell variable containing the path to the project's emb-bin directory so if you had some binary, `foo`, that you wanted to run on the target directory tree, you could use a line like:

```
$EMB_BIN/foo $TARGET_ROOT
```

The built in deploy variables are:

- `$PROJPATH` - path to the project directory
- `$EMB_BIN` - path to project's emb-bin directory
- `$TARGET_ROOT` - path to the root of the target directory tree

The output of all commands will be sent to a log box in a window for the deploy wizard.

### ***Package Deploy Configuration***

In addition to allowing shell script hooks to be called for each individual page, it is also possible to call specific sections of a package's `lbc` file for execution at specific points during deployment.

When the wizard looks for the action section of the `ldc` file for a page, it also looks for a magic string:

```
RUN FROM PACKAGE LBC %<section>
```

Where `section` may be any string. When the wizard sees this, it then checks every package's `lbc` file for the section specified and if it find's it, that section is executed with the output going to the same log window that the action output went to. This is probably best explained with an example.

Here is the above section that builds the filesystem, but including the magic string to invoke package level deploy configuration.



---

```

---- from deploy.ldc ----

%build_fs_action

if ["${REBUILD_FS}" = "0"] ; then option1="--nobuildfs"; fi
if ["${ADD_MERGE}" = "0"] ; then option2="--nomerge"; fi

/opt/Embedix/emb-bin/emb_mktime.py --projectdir ${PROJPATH}
${option1} ${option2}
if ["${ADD_KERNEL}" = "1"] ;
then
 mkdir -p ${PROJPATH}/tmp/rootfsdir/boot
 cp ${IMAGEPATH} ${PROJPATH}/tmp/rootfsdir/boot
fi

***RUN FROM PACKAGE LBC %deploy**

```

When the wizard runs the above action shell fragment, it will see the magic string. It will then make a list of all packages that are enabled for the current project. From there it will scan every package's lbc file (in local, board, generic order) looking for a "%deploy" section. For each deploy section that it finds, it will execute the shell fragment. The buildvars for that specific package will be exported to the environment during execution of the shell fragment so that this process can take advantage of configurability allowed by Target Wizard.

Here is a trivial example I used for testing for nano.

First, I modified nano's ecd to add some basic buildvars:

```

<GROUP Textprocessing>
 <GROUP Editor>
 <COMPONENT nano>
 DEFAULT_VALUE=0
 <HELP>
 nano (Nano's ANOther editor) - the small, capable,
 text editor formerly known as TIP (TIP Isn't Pico).
 It aims to emulate Pico as closely as possible while
 also offering a few enhancements.
 </HELP>
 KEEPLIST=/usr/bin/nano
 MIN_DYNAMIC_SIZE=0
 PROMPT=Include /usr/bin/nano?
 BUILD_REQUIRES=ncurses
 <REQUIRES>
 libncurses.so
 libc.so.6
 ld-linux.so.2
 </REQUIRES>
 SRPM=nano
 STATIC_SIZE=51966
 STORAGE_SIZE=132075
 TYPE=bool

 #added new stuff here
 <OPTION nano_variable>
 PROMPT=Set a nano string variable?
 BUILD_VARS=nano_var=$VALUE

```

---

```

 TYPE=string
 </OPTION>
 <OPTION nano_number_variable>
 PROMPT=Set a nano number variable?
 BUILD_VARS=nano_number_var=$VALUE
 TYPE=int
 </OPTION>
 #end of new stuff
</COMPONENT>
</GROUP>
</GROUP>

```

Notice that there are two variable options. Both of them can be set from Target Wizard. One is an integer and another is a string.

Then I added a section to nano's lbc file:

```

%deploy
echo "This is the deploy section for nano"
echo "the variable nano_var is ${nano_var}"

cat <<TXT > ${TARGET_ROOT}/nano.test
this is a test for nano.
The target root filesystem on host is at:
${TARGET_ROOT}
I hope that worked.
The string entered in Target Wizard is:
${nano_var}

The number string entered in Target Wizard is:
${nano_number_var}
TXT

```

All this does is write a file called nano.test to the root of the target filesystem that contains the buildvars in it. This example is somewhat trivial, but shows how the system could be used. With the functionality of running arbitrary sections from an lbc file, packages could contain more than just one shell fragment to run at different times.

For example, consider the following deploy.lbc file and package.lbc file:

```

---- ldc file portion ----
%build_fs_action
#stuff here
***RUN FROM PACKAGE LBC %makefs**

%make_cd_action
#stuff here for making a cd
***RUN FROM PACKAGE LBC %do_cd_stuff**

---- part of some packages lbc file ----
%makefs
#include stuff here to occur right after building the
#target file system

%do_cd_stuff
#include stuff here that you want to occur right after

```

---

#making a CD

## **Files**

The framework python file named `deploy_page.py` is located in the `/opt/embedix/lwiz1.0/usr/lib/python2.0/site-packages` directory

A set of example files can be found in the `examples` directory on the CD-ROM. Examine the `README` file there for instructions on using the example files for the `deploy hooks` feature.

## **Manual Deployment**

If you do not want to spend time on customizing the deployment wizard, it is possible to deploy the final image without using the `deploy wizard` at all. To do this, you will need to make the root filesystem and then move it to the location that you want to chroot to.

Make the root file system from the command line (or Makefile).

The tool is at:

```
/opt/Embedix/emb-bin/emb_mktimage.py
```

... and the usage is:

```
emb_mktimage.py --projectdir <project directory> [--nomerge]
```

By default the contents of the `merge` directory are put into the target image, but you can turn that off by using the `--nomerge` option. So to build the filesystem for a project called `intel`, you need to do this:

```
/opt/Embedix/emb-bin/emb_mktimage.py --projectdir
/home/chrisb/project/intel
```

This will always build the target filesystem with the root located at:

```
<project dir>/tmp/rootfsdir
```

You can either chroot to the `<project dir>/tmp/rootfsdir` or tar that directory up and move it to where you want.

```
tar czvf <project dir>/tmp/<new tar_ball.tgz> <project
dir>/tmp/rootfsdir
```

```
tar xzvf <project dir>/tmp/<new tar_ball.tgz> <your destination>
```

Note that this script doesn't update the kernel image in `rootfsdir`. That is, after the kernel is built, it is placed in `<project dir>/tmp` where you will need to `chmod` it (755) and then copy it into the filesystem each time the filesystem is rebuilt.

---

## Step 7: Creating a New BSP from the Project

Once you have customized your project to include new binaries, open source packages, kernels, deploy wizards, or other software elements, you might want to create a new Board Support Package. This board support package will be based on all of the elements within the project that you've just created. Packaging your project into a BSP will allow all SDK users to select your custom BSP, for use as the software baseline for an unlimited number of new projects/products.

### Using `export_bsp`

The `export_bsp` tool is useful for creating a bsp that includes the changes/modifications that you have made to the standard bsp distributed with the Embedix SDK. To use `export_bsp`, follow the guidelines below:

- Create a project in Target Wizard using the standard bsp.
  - Make any additions/modifications to the standard bsp. Make sure that your additions/modifications are in the:
    - ECD changes: `<project dir>/config-data/ecds/local`
    - LBC changes: `<project dir>/config-data/buildcontrol/local`
    - Package source changes: `<project dir>/Packages/local`
    - Precompiled binaries: `<project dir>/merge`
    - New saved state files (.sdf): `<project dir>`
    - Deployment script changes (`<board name>_emb_deploy`): `<project dir>/emb-bin`
- Build and test the changes you have made. Once the changes have been tested and are working the way that you want, use `export_bsp` to create a new bsp.
  - `cd /opt/Embedix/emb-bin/`
  - `./export_bsp --projectdir <project dir> --bspdir <new_bsp dir>`
    - For example: `./export_bsp --projectdir /home/user/project/x86 --bspdir /home/user/export_bsp-test-2.4` would create the new `/home/user/export_bsp-test-2.4` bsp directory. This directory will be populated with all of the changes you made to the standard bsp. NOTE: the '-2.x' at the end of the `<new_bsp dir>` name is required by `export_bsp`.
- `cd <new_bsp dir>/config-data/buildcontrol`
- edit the `bsp_config` and change the `"%name"` entry to reflect that it is your new bsp.
  - For example, you could change the `"%name"` entry from 'Generic x86 board' to 'My new x86 bsp'.
- `cd /home/user`
- Tar up the new bsp directory: `tar czvf <new_bsp dir>.tgz <new_bsp dir>`
- To use the newly created bsp simply untar the `<new_bsp dir>.tgz` in `/opt/Embedix/bsp`.
- Open Target Wizard and create a new project. Make sure to select 'My new x86 bsp' from the "Current Project Target Platform" list.

Note: If you have made changes to the GNU tools, these changes are not exported by the `export_bsp` utility. They must be copied and installed separately.

---

## Embedix Support

Support pages for Embedix SDK are maintained at: <http://members.lineo.com>

From these pages, you can access the following information:

- SDK Updates, Enhancements, and Bug Fixes
- SDK and Embedix Documentation
- Technical Information
- FAQs, How-Tos, etc.
- Linux Boot Loader Information

Additional support is available in accordance with the terms of individual support contracts. Contact [support@lineo.com](mailto:support@lineo.com) for further information.

---

## Notices

### Disclaimer

Embedix, 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. Embedix, 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.

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

### Trademarks

Lineo, Embedix and the stylized Lineo logo are registered trademarks of Embedix, Inc.

Linux is a registered trademark of Linus Torvalds

MontaVista is a trademark of MontaVista Software. MontaVista neither supports nor has participated in the development of this product.

TimeSys is a trademark of TimeSys, Inc. TimeSys neither supports nor has participated in the development of this product.

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

### Copyright Information

Copyright © 2002 Embedix, 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.

Embedix, Inc.

588 West 400 South

Suite 150

Lindon, UT 84042

USA

<http://www.lineo.com>