

TP2 Packager's Guide

Version 1.0, 2007 – Simon Edwards

Table of Contents

1	Purpose of Document	3
2	Introduction to Packaging	3
2.1	Package Format	3
3	Building a Package	4
3.1	Package Types	4
3.2	Package Configuration File Format	5
3.3	Generating the Package	5
4	Using Package Build Scripts	7
5	Discussion of Package Types	8
5.1	Binary Packages	8
5.2	Generic Packages	8
6	Using Package Install / Remove Scripts	9
6.1	When scripts can be run	9
6.2	Script Environment	10
6.3	Script Execution	11
6.3.1	Check-Install Script	11
6.3.2	Pre-Install Script	11
6.3.3	Post-Install Script	11
6.3.4	Pre-Remove Script	12
6.3.5	Post-Remove Script	12
6.4	First Boot Script	12
6.5	Script Definition in Package Configuration File	13
6.6	Typical Script Usage	13
7	Using Package "Check Install" Scripts	16
7.1	Purpose of Check Install Script	16
7.2	Script Environment	16
7.3	Supported Return Codes	17
7.4	Example Check Install Scripts	17
7.5	Understanding the Dependency Information	17
8	Package Dependency Management	19
8.1	Package Versions	19
8.2	Dependency Resolution	19
8.3	Using Alternative Suggested Repository	20
8.4	Dependency Specification	20
8.5	Package Incompatibles	21
9	Special File Handling	22
9.1	Configuration Files	22
9.2	Volatile Files	22
10	Appendix A: Configuration File Format	23
11	Appendix B: OS and Architecture Values	23

1 Purpose of Document

TP2 has been written to meet several requirements – one of which is ease of use. However, ease of use not only for the end user, but also for those generating packages. This document describes how packages are generated – and also how that process can be integrated into the CM2 code management system.

2 Introduction to Packaging

Rather than attempt to describe packaging in great detail to approach here is to describe the generation of a simple package, and then describe additional features that can be used. This approach should allow most packages to be built with the minimum of reading and preparation.

2.1 Package Format

Before describing the process of generating a package it is useful to understand how the contents of a package are organised. At worst this is useful background information – though more likely it will enable developers to extend the existing packaging facilities or to customize them in some way.

A package is simply a Gzipped-tar archive – indeed this should be obvious to those that have manually installed the toolset in the past. The reason for using this format is that both Gzip and Tar are very common and available for all target platforms.

The packaging overhead is very small – simply one file. The files that are deployed by the package are all in relative format – since TP2 requires that it must be possible to install the package into multiple namespaces on the same machine simultaneously.

If a package was to deploy the following files:

```
bin/program1
bin/program2
share/man/man1/program1.1
share/man/man1/program2.1
```

then the files in the gzipped-tar archive would be:

```
!!..config..!!
bin/program1
bin/program2
share/man/man1/program1.1
share/man/man1/program2.1
```

Hence, even if TP2 is not installed on a machine getting at the contents of a file is very straightforward. Typically the process would be:

```
# mkdir /tmp/tmpdir
# cd /tmp/tmpdir
# gunzip -c package.tp2 | tar xvf -
```

This is indeed the process that occurs during the initial installation of TP2 – whether from the automated “bootstrap” script, or via the manual installation.

3 Building a Package

Now that the contents of the configuration file have been outlined a sample package can be built as using the example files shown previously.

3.1 Package Types

Since TP2 is a generic packaging format that has been designed to be suitable for all UNIX-like environments, a package can be similarly flexible. The following table shows the available types of package that can be generated:

Type	Purpose
Generic	<p>If the contents of a package do not contain binary executables and can be deployed to any platform and any machine architecture, then that package type is known as "generic" – it can be installed on any machine that can use the TP2 packaging tools.</p> <p>This type of package is particularly suitable for handling packages that contain programs that are text-based rather than binary. For example suites of Perl, shell or Python scripts.</p> <p>This package type can include binary files – though of course the format should be well defined to ensure that they are compatible across 32 and 64 bit platforms and different CPU endianness.</p>
OS Specific	<p>This package has been designed to work on a specific OS version – such as Linux or Solaris. It can not be installed on other UNIX-like variants.</p> <p>This package type is not architecture specific – it does not contain files that are not compatible across different CPU types. An example of such a package might be a series of shell scripts designed for a particular OS variant.</p> <p>The later sections of the document describe the settings to use for each OS variant.</p>
Architecture Specific	<p>An architecture specific package contains files that are only suitable for the specified CPU type – though might be able to be used on any operating system variant for that architecture.</p> <p>An example of such a package type might be a series of data files for a sample database – that contains endian information, but will work across multiple operating systems supported on that chip type – Linux and BSD for example.</p> <p>Sections later in this document contain information on the various settings that can be used to specify architecture.</p>
OS and Architecture Specific	<p>The other possible package type is one that determines the OS variant and the architecture that the package can be installed on. This is quite a common method of distributing packages – for example it is common for binary packages to deliver files for a 32bit Linux variant, whilst another build of the package is provided for a 64bit Linux variant.</p>

3.2 Package Configuration File Format

A package is generated by copying the contents that are meant to make up a package to a temporary directory, and then using the "tp2pkg" package along with a configuration file to generate the package.

The contents of the package are generated from a *temporary* directory because the packaging process may alter the contents of the directory. There is a single configuration file that drives the package generation. This file is a simple XML file. Again consider a package delivering the following which are considered to be generic:

```
bin/program1
bin/program2
share/man/man1/program1.1
share/man/man1/program2.1
```

In this instance the package generation file might consider of just the following contents:

```
<?xml version="1.0" standalone="yes"?>
<tp2package>
  <name>example_pkg</name>
  <description>An example test package</description>
  <os>GENERIC</os>
  <architecture>GENERIC</architecture>
  <version>1.1.0</version>
  <files>
    <file perms="755" owner="root" group="sys">bin/program1</file>
    <file perms="755" owner="root" group="sys">bin/program2</file>
    <file perms="444" owner="root" group="sys">share/man/man1/program1.1</file>
    <file perms="444" owner="root" group="sys">share/man/man1/program2.1</file>
  </files>
</tp2package>
```

There are several points to notice about this configuration file:

- Each and every file that the package deploys must be indicating, along with permissions [in octal format], owner and group of the files.
- The directories do not need to be specified – they will be generated automatically as the installed user, group and with the default umask settings. Of course they can be specified if necessary.
- The "os" and "architecture" values are given the special value "GENERIC" to ensure a package that is both OS and architecture neutral is generated.

3.3 Generating the Package

By convention when a package is generated the package configuration file is kept in a directory called "pkg". The configuration file can be called anything of course, though "conf" or "pkgconf" are common names.

To generate a package the "tp2pkg" command is run from the top-level directory of the temporary copy of the package contents. For this particular example the temporary directory "/tmp/tmpdir" thus has the following files/directories.

TP2: Packager's Guide

```
bin/program1
bin/program2
share/man/man1/program1.1
share/man/man1/program2.1
pkg/conf
```

To generate a package use the following commands:

```
$ cd /tmp/tmpdir
$ tp2 pkg --config pkg/conf --verbose
```

The “--verbose” option is commonly used since it provides feedback on the packaging process to the current “Standard Out” device – which is typically the terminal. In this instance the output should would appear similar to the following:

```
Log : Well-formed XML in "pkg/conf" - continuing.
Log : Validated XML in "pkg/conf" - continuing.
Log : Packaged will be spooled to: /tmp/example_pkg+1.0.0.tp2
Log : Package generated successfully.
```

By default the generated package is copied to “/tmp” – though that can be changed through the use of command line options. Also notice the name of the package generated – it defaults to the following format for “generic” packages:

```
<pkgname>+<pkgversion>.tp2
```

If the specified package name already exists it will not be over-written – an error will be shown instead.

4 Using Package Build Scripts

Now that a simple package has been generated some more useful features that TP2 offers can be explained. The first is that it **does not** support the concept of a package "build" script. To consider this firstly imagine the sample package was actually built from the following directory structure before being copied to the temporary area:

```
bin/program1
bin/program2
src/Makefile
src/program1.c
src/program2.c
share/man/man1/program1.1
share/man/man2/program2.1
```

In this example the intention is to distribute the contents of the directories apart from the contents of the "src" directory. This raises several points:

- In this case "program1" and "program2" are likely to be specific to this architecture and this operating environment – rather than generic.
- The "Makefile" is a method of generating the "program1" and "program2" binaries and should be called just before they are packaged.

The package configuration file in this case would appear similar to the following – the differences from the first example are quite small:

```
<?xml version="1.0" standalone="yes"?>
<tp2package>
  <name>example_pkg</name>
  <description>An example test package</description>
  <
  <version>1.1.0</version>
  <files>
    <file perms="755" owner="root" group="sys">bin/program1</file>
    <file perms="755" owner="root" group="sys">bin/program2</file>
    <file perms="444" owner="root" group="sys">share/man/man1/program1.1</file>
    <file perms="444" owner="root" group="sys">share/man/man1/program2.1</file>
  </files>
</tp2package>
```

Notice that the architecture and OS entries are not present and in this case they will default to the current OS and architecture. Hence if the above contents are copied to a temporary directory, then the following commands might be called to generate the package:

```
$ cd /tmp/tmpdir
$ cd src && make
$ cd ..
tp2 pkg --config pkg/conf --verbose
```

Of course this can be simplified if the "CM2" code management suite is in use since it supports TP2 package generation including automated calling of build scripts as part of package generation. In this case the generated package name will be:

```
example_pkg+Linux+i386.tp2
```

5 Discussion of Package Types

5.1 Binary Packages

The above example of using a build script raises an interesting point; should "binary" packages [or more specifically architecture/OS restricted packages] be used? Consider the advantages of such packages:

- *Ease of installation* – with a binary package the user does not need to have an environment to compile the source – typically a package can simply be installed without making too many demands on the target environment.
- *Speed of installation* – if a lot of packages or a package containing large programs needs to be installed, simply installing pre-compiled binaries can result in the installation process taking seconds rather than hours [to compile the source].
- *Package confidence* – if a user have a package that is tied to a OS variant and architecture they can be confident that the contents of the package are geared toward their environment and should work simply work.

However the disadvantages are also significant:

- *Limiting Target Audience* – if you develop on BSD and make only OS and/or architecture specific packages others on different platforms can not make use of your software.
- *Loss of Binary Optimisation* – If you have a package that is designed to run on a series of CPU architectures you must ensure the code is not optimised for newer versions of the architecture – otherwise you limit the target audience for the package further.
- *Management Overhead* - if a package is defined for each architecture and OS variant then the developers must make the effort to generate many packages for each set of architectures and/or OS variants they intend to support.

5.2 Generic Packages

The advantages of generic packages are essentially the opposite of the advantages and disadvantages of architecture or OS specific packages. Such packages make sense when;

- The software is a series of scripts rather than binary programs – think collections of shell, Perl, Python, Tcl or other "scripted" languages.
- The executables that the package intends to deliver can be built quickly and easily without a significant number of other dependences being required on the target machine.
- The software makes use of OS or architecture facilities that are native to certain platforms, and thus a native compile is required.

6 Using Package Install / Remove Scripts

For most of the most basic type of packages simply installing a series of files will be enough – but often it is useful to be able to execute a script when a package is installed – or even removed.

6.1 When scripts can be run

The TP2 software suite is able to run scripts when packages are installed, removed or even when the machine is first rebooted following a package installation. The following table summarises the supported scripts:

LABEL	Purpose
CHECKINSTALL	This script is run even before a package is copied from a repository! It is a script that resides in the repository index file and is used to check the suitability of the target platform for package installation.
PREINSTALL	This script is run just prior to installing the specified package. The fact that this script is being executed means that the package will be installed - though if this script fails with a certain return code it can still abort the installation.
POSTINSTALL	Once the package has successfully installed all files this script is run to perform any post-configuration, such as creating directories and default file entries, for example. Again the return code can be used to indicate or even abort the package installation even at this late stage.
PREREMOVE	When a package is about to be removed this script is run. The return code can be used to determine whether to continue with the remove process or not.
POSTREMOVE	Following the package removal this script can be used to perform any remaining clear-up - for example removing logs or directories created as part of the "POSTINSTALL" script.
FIRSTBOOT	When this script is installed the first time the machine is rebooted after package installation this script will be run. If necessary this script can run once, run every boot until it works successfully, or run every time whether it succeeds or not.

It should be noted that these are "scripts" – they are expected to be clear-text that can be readily executed on the expected target platforms. In almost all cases these will be shell scripts – and it is recommended that they are simply Bourne shell compatible scripts using the "/sbin/sh" executable.

The scripts are run as the installer of the package - which in most cases will be the namespace owner. The "firstboot" script will always run as the owner of the namespace even though that part of the boot process is actually controlled by "root".

6.2 Script Environment

For each of the four scripts that can be run the following environment variables will be set and can be made use of in any script.

Variable	Purpose
PKGROOT	The name space root directory, such as <code>"/opt"</code> or even <code>"/home/sedwards"</code> .
PKGSPACE	The name of the current namespace the package is being installed to.
PKGNAME	The name of the package being installed.
PKGVERSION	The version number of the package being installed.
PKGACTION	The action that the script has been called for – will be either <code>"PREINSTALL"</code> , <code>"POSTINSTALL"</code> , <code>"PREREMOVE"</code> , <code>"POSTREMOVE"</code> and <code>"CHECKINSTALL"</code> .

The Check-install script is a powerful mechanism for dynamically altering the list of dependencies that a package must install on a particular platform. Hence to achieve this there

Variable	Purpose
PKGDEPENDS	A base list of dependencies for this package - base separated.
PKGMSGFILE	This is a file that the check install script can write to which will affect the actual package installation. See Appendix C for examples.

If a package is being installed some additional variables are set for the pre-install and post-install scripts.

Variable	Purpose
PKGVERSION_PREVIOUS	This is set if the package is already installed. It is the version number of the package that is being over-written.
PKGINSTALL_ACTION	This is used to indicate the type of installation that is being performed: <ul style="list-style-type: none"> • install - The package is currently not installed. • downgrade - The package is currently installed at a greater version number than the package is currently being installed at. • upgrade - The package is currently installed and the new version being installed is at a higher level. • reinstall - The same version of the package is already installed.
PKGSINSTALLED	This provides a summary of all the existing packages that are installed in the namespace before any installations involving the current install session takes place.

When the script is run it is done so from the package root directory, so initially the current working directory is the root directory for the namespace.

6.3 Script Execution

Each of the available scripts will now be described in detail. It should be noted that the return code the script generates is very important – it will determine whether the user intended action [whether it is package installation or removal] will be completed or aborted.

6.3.1 Check-Install Script

This script is run when a particular version of the package has been chosen for installation - but even prior to the package being retrieved from the repository. Typically such scripts are used to ensure that packages can de-select themselves from a package installation if they believe they are not suitable for the intended host.

Examples of Check install scripts and how to alter the dependencies in a dynamic fashion can be found in the next section.

The return code issued by the script is important, and must be one of the following:

- 0 - The script ran successfully.
- 1 - A warning return code has been issued - though installation should continue.
- 2 - A fatal error has been returned - the installation process will be aborted.
- 3 - The script has indicated that this package should remove itself from the list of packages to install.

6.3.2 Pre-Install Script

The pre-install script is run just prior to loading the files that are to be installed as part of the package. This script is run if the package is defined as being suitable to be installed. Again the standard input, output and standard error whilst this script is run are unaltered - so works well when running from the command line.

The return code issued by the script is important, and must be one of the following:

- 0 - The pre-install script ran successfully
- 1 - The pre-install script completed with warnings
- 2 - The pre-install script aborted - abort software installs (unless forced)

6.3.3 Post-Install Script

The post-install script gets run following all file deployments and directory creations for the package. The very fact that it has been run indicates the complete contents of the package has been successfully installed. Unlike many package managers the post-installation script **impacts whether the package is installed successfully or not**. The return codes are the same as with the pre-install:

- 0 - The post-install script ran successfully
- 1 - The post-install script completed with warnings
- 2 - The post-install script aborted - abort software installs (unless forced)

The environment variables set are also the same as for the pre-install script. Hence if the script "aborts" all software installations will "roll back" to the previous configuration. This means that if the script returns "2" all the files that the package deployed will be removed, and all directories [if empty] that were created will also be removed. If the package over-wrote any existing files during the installation the previous contents will be put back in place.

6.3.4 Pre-Remove Script

This script is run prior to starting the removal of a package. The fact that this script is running indicates that the package is going to be removed (either because of an explicit reference in a command), or implicitly due to dependency issues.

As with the installation scripts the following return codes are expected from this program, which must be a normal script:

- 0 - The pre-remove script ran successfully
- 1 - The pre-remove script completed with warnings
- 2 - The pre-remove script aborted - abort software removal (unless forced)

The environment variables set are also the same as for the pre-install script. Hence if the script "aborts" all software installations will "roll back" to the previous (installed) configuration.

Because this script is run before the package contents are removed the script is able to call any of the files that the package deployed to check the environment if necessary. Remember that the program or script referenced will need to be referenced absolutely via the package root directory or relative to the current directory.

6.3.5 Post-Remove Script

Once all files have been removed (directories might still exist if other packages deployed into them or other files have been created in them) this script is called – if defined as part of the package. The return codes are the same as the "pre-install" script as defined in the previous section.

Please note that if the post-remove script fails and the "force" option has not been specified then the package will re-instate itself to an "installed" state if at all possible.

Only when this script has completed will the files finally be removed from the file system space.

6.4 First Boot Script

This script is usually executed just once when the machine is first rebooted following the package installation. It is often used to ensure all other version information is in place or to set flags to indicate that a reboot has been completed and the package can now be used [useful if the package influences the machine kernel in some way].

The return code is important and in this respect the following values affect what happens to the first boot script:

- 0 - The first boot script has completed successfully and should be removed from subsequent reboots.
- 1 - The first boot script failed - but it should still be removed from subsequent reboots.

- 2 - The first boot script has completed successfully, but it should be kept for the next time the first boot process is used.
- 3 - The first boot script failed and the script should be kept to run again when the first boot process next occurs.

6.5 Script Definition in Package Configuration File

All scripts are optional, but if they are required an entry must be present in the package configuration file. Taking the previous example package configuration entries for all of the above have been added and are shown in bold below.

```
<?xml version="1.0" standalone="yes"?>
<tp2package>
  <name>example_pkg</name>
  <description>An example test package</description>
  <os>GENERIC</os>
  <architecture>GENERIC</architecture>
  <version>1.1.0</version>
  <checkinstall>pkg/checkinstall</checkinstall>
  <preremove>pkg/preremove</preremove>
  <postremove>pkg/postremove</postremove>
  <preinstall>pkg/preinst</preinstall>
  <postinstall>pkg/postinstall</postinstall>
  <firstboot>pkg/firstboot</firstboot>
  <files>
    <file perms="755" owner="root" group="sys">bin/program1</file>
    <file perms="755" owner="root" group="sys">bin/program2</file>
    <file perms="444" owner="root" group="sys">share/man/man1/program1.1</file>
    <file perms="444" owner="root" group="sys">share/man/man1/program2.1</file>
  </files>
</tp2package>
```

The names of the scripts used does not matter – however they should be clear-text and not binary in nature. The scripts should also be self-contained – they should not call other scripts as part of the installation – unless these are deployed as part of the package – though that is only possible for pre-remove and post-install scripts.

Of course none of these are required and if not needed can be commented out or left out of the file completely.

The convention when building a package is to call the script names the name of the action – such as “preremove” – though the name does not matter. It is also common to keep the scripts a sub-directory called “pkg” – which is the location often used for the package configuration file itself.

As stated all scripts are optional and the ordering of the entries does not matter. The exact contents of the script specified will be rolled into the package’s “!..config..!” file and will be extracted and executed when required as part of package installation or removal.

6.6 Typical Script Usage

Depending on the size and complexity of the package that is to be installed the scripts can be used for several reasons.

Script	Usage
CHECKINSTALL	Validate the environment is suitable for the package and add or remove dependencies based on the machine configuration.

Script	Usage
PREINSTALL	May check to see if an existing version of the package is already installed, and if so may stop existing programs.
POSTINSTALL	A post install script is often used to configure a package – for example if a package deploys source files it might call another script deployed by the package to build the binaries for the current installation.
PREREMOVE	As with a pre-install script this may ensure that any daemons that might be running for the deployed package are stopped.
POSTREMOVE	If a program generated binary programs as part of the post-installation script the post-remove script should be written to remove such files.
FIRSTBOOT	Check whether kernel changes put forward during the package installation have been completed successfully and thus perform some form of package administration.

There are several ways of checking to see if an existing package is installed. The most common way is simply checking for the existence of a file that the package might have deployed. Consider the following script:

```
#!/bin/sh
if [ -f $PKGROOT/bin/daemon ]
then
    $PKGROOT/bin/daemon stop
fi
exit 0
```

The above might appear as a pre-install script. It checks to see if a daemon that is about to be deployed exists and if so stops it. This is a common way of ensuring packages that contain binaries are able to deploy all there files [instead of not deploying the file since the contents of the file are "busy".

Also the above script could be used equally well as a pre-remove script – ensuring that the package binaries are not in used and thus can be easily removed.

A simple post-installation script might be:

```
#!/bin/sh
cd $PKGROOT/src
make || exit 2
make install || exit 2
exit 0
```

The above calls expects the package to deploy a "Makefile" in the "src" directory to build and install files. Of course the Makefile should be written to ensure it deploys all files into the \$PKGROOT directory tree.

Note that a post-installation script can not make use of any of the files that the package deployed – unless they were generated by the post-installation script of course. Hence a post-remove script may explicitly have to do much work itself, for example:

TP2: Packager's Guide

```
#!/bin/sh
rm -f $PKGROOT/bin/program1
rm -f $PKGROOT/bin/program2
exit 0
```

Future enhancements due in 2007 to TP2 will include the ability for post-installation scripts to "Register" other files into the package meaning that post-removal scripts will be able to leave the removal of such generated files to the package management software itself.

7 Using Package “Check Install” Scripts

7.1 Purpose of Check Install Script

Although package installation and removal scripts have been discussed in the previous section there is one other script that can be used as package installation – this is called the “check install” script.

This script is different to any of the install or remove scripts since it does not only appear as part of the package – but is also copied into a package repository index. Like the other scripts this must be clear-text and should be a Bourne shell compatible script to ensure it can execute on any of the target platforms.

The purpose of the check installation script is to check the environment into which a script is supposed to be installed and then indicate via the return code as whether to allow or refuse the package installation.

This script is run before any of the contents of the package are even downloaded, never mind installed. This is very important – since it allows the package developers to ensure that potential package installers know without having to download the package itself – which can be frustrating if downloading from a remote repository over a slow link.

7.2 Script Environment

The following environment variables are available for the “Check Install” script;

Variable	Purpose
PKGROOT	The name space root directory, such as “/home/sedwards”.
PKGSPACE	The name of the current namespace the package is being installed to.
PKGNAME	The name of the package being installed.
PKGVERSION	The version number of the package being installed.
PKGACTION	The action that the script has been called for – will be either “PREINSTALL”, “POSTINSTALL”, “PREREMOVE”, “POSTREMOVE” and “CHECKINSTALL”.
PKGMSGFILE	The name of a temporary file that this script can write messages to, to pass back information (over and above the return code), to the main package installation.
PKGDEPENDS	A space-separated list of dependencies recommended for this package.

If the check-install script writes output to standard output it will appear on the terminal or be shown in the display or the installation GUI window.

7.3 Supported Return Codes

The check install script is expected to return one of the following return codes, though others are simply ignored:

- 0 - The check install script ran successfully
- 1 - The check install script completed with warnings
- 2 - The check install script aborted - abort software installs (unless forced)
- 3 - The check install indicates this package is not needed

Unless the return code is 2 or 3 then the package will be considered for installation. When a return code of 2 is given, then unless the "--force" argument is specified the installation of all packages will be aborted.

A return code of 3 is unusual - it indicates that the package has removed itself from the list of packages to install. This allows a check install script to scan the environment and not install itself (or any of its necessary dependents) if certain conditions are true.

7.4 Example Check Install Scripts

A common use of check-install scripts is to indicate that the package in question is not suitable for the chosen platform. This approach allows the package to be made generic but the check-install script to issue a warning if an install is attempted on an unknown platform. Consider the following example:

```
#!/bin/sh
if [ -n "$PKG_FORCE_INSTALL" ] && [ $PKG_FORCE_INSTALL -eq 1 ]
then
    exit 0
fi
if [ `uname` != "Linux" ]
then
    echo "Package has only previously been installed on Linux."
    echo "To force installation on this OS set PKG_FORCE_INSTALL to 1."
    exit 2
fi
```

Or consider the following:

```
#!/bin/sh
if [ -x "$PKGROOT/bin/programX" ]
then
    echo "Found programX - indicating this package is not needed."
    exit 3
fi
exit 0
```

This is a useful way of checking to see if this package is no longer needed since a program that provides the features this package provides is already installed.

7.5 Understanding the Dependency Information

When the check install script is run the PKGDEPENDS environment variable contains a list of dependencies that the package indicated was being set when it was built [see the next section for more details on dependencies].

The contents of this variable, if not empty will be in the format of one or more dependencies, space separated. The format of each being:

```
dep1[,minver=N.N.N,maxver=N.N.N]
```

The check-install script may or may not use this information. The list does not indicate what packages are currently installed – the main installer will use that information to work out what else must be installed if the check-install script actually completes successfully.

What is more interesting is that this list of dependencies can be affected by the check-install script. It does this by writing one or more lines to the file name \$PKGMSGFILE. Consider the following example check-install script:

```
#!/bin/sh
if [ ! -x $PKGROOT/bin/programX ]
then
    echo ">DEPENDS pkgX.minver=1.0.0" >>$PKGMSGFILE
fi
exit 0
```

The above check-install script checks to see if a program is installed in the namespace and if not sends a message back to the packaging software that package "pkgX" is a dependency before this package can be installed.

The lines in the \$PKGMSGFILE can be in any of the following formats. Other lines in other formats are ignored.

Action	Purpose
=DEPENDS [dep1[,minver=N.N.N,maxver=N.N.N] ...]	The complete list of original dependencies is replaced by this list. Must only occur once in the file.
>DEPENDS [dep1[,minver=N.N.N,maxver=N.N.N] ...]	If the package already exists in the list of dependencies it will be altered to the versions required, (defaulting minimum to 0.0.1 and maximum to 999999.9.9) if not given. If the package is not currently in the list it will be added to it. Can occur multiple times.
-DEPENDS [dep1 ...]	Removes the specified packages from the list of dependencies (if they are currently included). Can occur multiple times.
-DEPENDS *	Remove all dependencies from the list.

8 Package Dependency Management

The TP2 packaging suite fully supports dependencies. Before these can be used as part of any generated packages a little background on how dependencies are resolved in TP2 is useful.

8.1 Package Versions

TP2 tries not to enforce a particular format to the packages that it can deploy. However a common convention is to use version numbers in the following format:

```
Major.Minor.Increment
```

For example the first major release might be 1.0.0, the first bug release following this "1.0.1", whilst the first package with incremental improvements beyond this is "1.1.0".

However TP2 can support versions in the following format

```
Version ::= <RelNum>[.<RelNum>]+
RelNum ::= <0-9>+
```

That is the following could be used if required:

```
1
1.2
1.200.2
1.2.3.4.5
```

However the recommendation is to stick with the "Major.Minor.Increment" format if possible. If you have the same packages with different formats of version numbers care must be taken since the comparisons for which release is newer might not work as you expect. Consider the following examples:

Version 1	Version 2	Result
1.0.0	1.0.1	Version 2 is the latest version.
1.0	2	Version 2 is the latest version.
0.2000	0.3	Version 1 is the latest version.
999.02	1000	Version 2 is the latest version.
0.0.0.0.1	0.0.0.2.0	Version 2 is the latest version.

The 3rd example works as it does since each portion of the version is treated as a separate number – here "2000" is being compared to "3".

8.2 Dependency Resolution

Dependencies in TP2 are enforced – it does not allow a package to be installed if the dependencies are not currently installed – or can not be found to install.

Unlike some package managers when a dependency required for the software is not installed it will search the repositories defined as part of the current "TP2_PATH" for a package of the

required level to meet the dependency. This process is recursive and so installation of a single package may result in the actual installation of (in theory) hundreds of packages.

All the dependencies for a certain package do not need to reside in the same repository - they can be spread over all configured repositories. It is also possible for a package to suggest where its dependencies might be found – see the next section for details.

8.3 Using Alternative Suggested Repository

Most package management suites support the ability to handle multiple repositories, and TP2 is the same. When searching for a package it will make use of the current setting for "TP2_PATH" to search for the package. This environment variable can consist of any number of repositories that are searched in order.

However it should be remembered that the TP2 installation routine [command line] also has the option "--repos" to allow additional repositories to be specified as possible sources of installs at any time [See "TP2 Administration Guide" for details].

One common problem that some package management suites face is that when a package is installed which requires further obscure dependencies this current set of repositories might not be suitable and will require the user to search for alternative repositories to use.

TP2 allows this problem to be overcome by allowing a package to specify alternative repositories – which can be used to help satisfy dependencies. Alternative repositories can be optionally specified by adding the following line in a package configuration file:

```
<altrepos>WWW:siteA/tp2packages WWW:siteB/packages</altrepos>
```

The "alterpos" setting can contain multiple repositories – each must be white space separated. If a dependency requested for a package can not be found in the list of standard repositories these repositories will also be checked.

The alternative repository settings only apply to direct dependents of a package – if any of those packages require dependencies their own "altrepos" settings [if defined] will be used if necessary.

8.4 Dependency Specification

A separate section in the package configuration file is used for dependencies – as shown in bold below:

```
<?xml version="1.0" standalone="yes"?>
<tp2package>
  <name>example_pkg</name>
  <description>An example test package</description>
  <os>GENERIC</os>
  <architecture>GENERIC</architecture>
  <version>1.1.0</version>
  <dependencies>
    <pkg>pkg1</pkg>
  </dependencies>
  <files>
    <file perms="755" owner="root" group="sys">bin/program1</file>
    <file perms="755" owner="root" group="sys">bin/program2</file>
```

```
<file perms="444" owner="root" group="sys">share/man/man1/program1.1</file>
<file perms="444" owner="root" group="sys">share/man/man1/program2.1</file>
</files>
</tp2package>
```

Multiple packages can be specified as dependencies – and it is also possible to specify dependencies in a range of formats, for example:

```
<dependencies>
  <pkg>pkg1</pkg>
  <pkg minver="0.9.0">pkg2</pkg>
  <pkg maxver="1.9.9">pkg3</pkg>
  <pkg minver="1.0.0" maxver="1.9.9">pkg4</pkg>
</dependencies>
```

The above section indicates the following dependencies:

- *pkg1* – any version can be installed.
- *pkg2* – a version of at least "0.9.0" must be installed. If a lower version is already installed it will be upgraded to at least this version before the actual package requested to install will be installed.
- *pkg3* – The package requires that at most version "1.9.9" if package "pkg3" is installed. If it is not installed a version of at most this version must be found and installed as part of the dependencies. If a later version is already installed the installation of this package will fail with a suitable error message.
- *pkg4* – If a version less than "1.0.0" is already installed it will be upgraded to at least "1.0.0". If no pkg4 is currently installed at least "1.0.0" but less than "1.9.9" will be installed. If a version of pkg4 is already installed in the range of "1.0.0" to "1.9.9" no action is taken. Finally if pkg4 is already installed, but at a level greater than "1.9.9" then the package installation will abort with a suitable error.

There are no limits to the number of dependencies that a package can specify. As stated package dependency management is fully recursive – if one of the dependencies specifies other dependencies these will be met as well, otherwise the package installation requested will not occur.

8.5 Package Incompatibles

As well as being able to specify package dependencies it is also possible to indicate incompatible packages. When an incompatible package is installed this package can not be installed at the same time.

To specify incompatible packages the package configuration file can have a section similar to the following:

```
<incompatibles>
  <pkg>fred</pkg>
</incompatibles>
```

When a user attempts to install a package which currently has an incompatible package already installed they will receive an error message similar to the following:

```
Error: Package "example_pkg" is incompatible with installed package "fred".
```

9 Special File Handling

Apart from the deployment of standard files and directories, TP2 is able to handle two special instances; configuration files and volatile files. Each is of these type of files are handled in a special way as this section describes.

9.1 Configuration Files

Configuration files are files that contain information that determine how the other programs in the package might be used. Why configuration files are dealt with differently compared to normal files is because of the way they are installed if the package is being upgraded and an existing configuration file found.

When a package is being upgraded the contents of the configuration file are compared to what was originally installed. If the contents of the package have been altered [i.e. modified by the user], then the version from the new package **does not** over-write the current one – instead it is written to a ".new" file to allow the administrator to compare the new one whether their own customised one. If the contents of the package are the same as was originally installed then the new one overwrites the existing one directly.

The same concept takes place if a package is removed; if the administrator has changed a configuration file it is simply renamed to a ".saved" version, but if they have not altered it is simply removed.

To indicate that a file is a configuration file the "configfile" attribute should be set, as shown in the following example [shown in bold]:

```
<files>
<file perms="755" owner="root" group="sys">bin/program1</file>
<file perms="755" owner="root" group="sys">bin/program2</file>
<file perms="444" owner="root" group="sys">share/man/man1/program1.1</file>
<file perms="644" owner="root" group="sys"
  configfile="1">cfg/pkg.conf</file>
</files>
```

9.2 Volatile Files

A volatile file is handled in much the same way as normally installed files. The major difference is that a checksum is not kept for volatile files – hence if the contents are changed the package verification tool will not indicate this is an error.

To indicate that a file is volatile the "volatile" attribute should be set as shown in the following example in bold:

```
<files>
<file perms="755" owner="root" group="sys">bin/program1</file>
<file perms="755" owner="root" group="sys">bin/program2</file>
<file perms="444" owner="root" group="sys">share/man/man1/program1.1</file>
<file perms="644" owner="root" group="sys"
  volatile="1">var/example_log</file>
</files>
```

10 Appendix A: Configuration File Format

11 Appendix B: OS and Architecture Values