

## **TP2 Packaging Packager's Guide**

### **Synopsis**

This document covers the relevant toolset that can be used to create and manage "tp2" packages – bundles of code that can be installed, removed, checked and updated in a controlled manner". It is aimed at administrators of the environment – developers are expected to have little input in the packaging.

## Revision History

Version	Date	Author	Changes
1.0	May, 2007.	Simon Edwards	Original version
1.4	April, 2011	Simon Edwards	Complete document review, fixes and refresh.
1.5	December, 2012	Simon Edwards	Updated to cover 1.4.0 release.

## Table of Contents

1	Introduction .....	4
1.1	Purpose of Document .....	4
1.2	Target Environment Requirements .....	4
2	Basic Package Format.....	5
2.1	The “!..config..!” File Contents .....	6
3	Building a Package .....	7
3.1	Package Types .....	7
3.2	Package Configuration File Format .....	8
3.2.1	Implicit or Explicit Directories .....	8
3.3	Generating the Package.....	9
4	Using Package Build Scripts .....	10
4.1	Use of Binary Packages.....	11
4.2	Use of Generic Packages.....	11
5	Using Package Install / Remove Scripts .....	12
5.1	When scripts can be run .....	12
5.2	Script Environment.....	13
5.3	Script Execution .....	14
5.3.1	Pre-Install Script.....	14
5.3.2	Post-Install Script .....	14
5.3.3	Pre-Remove Script.....	15
5.3.4	Post-Remove Script.....	15
5.3.5	Final-Install Script.....	16
5.4	Script Definition in Package Configuration File.....	17
5.5	Typical Script Usage .....	18
6	Usage Package Verify Scripts .....	20
6.1	Purpose of Verify Scripts.....	20
6.2	Script Environment.....	20
7	Using Package “Check Install” Scripts .....	21
7.1	Purpose of Check Install Script.....	21
7.2	Script Environment.....	21
7.3	Supported Return Codes.....	22
7.4	Example Check Install Scripts.....	22
7.5	Understanding the Dependency Information .....	23
8	Package Dependency Management .....	24
8.1	Understanding Package Versions .....	24
8.2	Dependency Resolution .....	25
8.3	Using Alternative Suggested Repository .....	25
8.4	Dependency Specification .....	26
8.5	Package Incompatibles .....	27
9	Text File Support.....	28
10	Package Deployment Filtering Support .....	29
11	Working with Package Bundles .....	30
11.1	What are Bundles? .....	30
11.2	Building a Bundle .....	30
11.3	Installation and use of Bundles .....	31
12	Special File Handling .....	33
12.1	Configuration Files.....	33
12.2	Volatile Files.....	33

# 1 Introduction

## 1.1 *Purpose of Document*

TP2 (or TruePackager2) is a package management system written for Linux/UNIX - and limited functionality on Windows too. There are many different packaging formats available for UNIX/Linux, though TP2 does offer several unique features.

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.

This document is meant to help developers understand the facilities available to create packages rather than manage an infrastructure of packages – that detail can be found in the “Administrators Guide” document.

## 1.2 *Target Environment Requirements*

The target environment to receive a TP2 package must have the TP2 software installed. This consists of a small set of Perl programs and modules that depend on. The Perl code has been kept as simple as possible; meaning any modern (from 5.6 and above) version of Perl interpreter should work with the code.

The actual Perl module dependencies for TP2 are described in the Administrators Guide – but are fairly common modules and where possible are made optional.

## 2 Basic 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 if deemed appropriate.

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. This format also means that native Perl libraries can be used if present, but also that default system binaries can be utilised as fall backs if necessary.

The packaging overhead is very small – simply one file and any digital signatures associated with the package. The files that are deployed by the package are all in relative path format – since a key feature of TP2 is the ability to install the same package into multiple namespaces on the same machine simultaneously.

Consider the following list of files that a package is to deploy:

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

Assuming the package is not digitally signed then the files in the gzipped-tar archive would be as follows:

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

The package has been signed by one or more people the contents will instead look like the following:

```
-rw-r--r-- pete/pete      67819 2011-04-23 16:35 ./__TP2_PACKAGE__
-rw-r--r-- pete/pete      154 2011-04-23 16:36 ./__TP2_SIGNATURE__
```

In this case notice that the “\_\_TP2\_SIGNATURE\_\_” is just a text file containing the signatures, for example:

```
-----BEGIN SIGNATURE-----
MC0CFQCJBou6z19JvBS4QvS2OBkzhW1yYgIUmaEoD90vf/300yFIfSkrVfu9BL4=
-----END SIGNATURE-----
Name: Fred
Email: fred@linuxha.net
-----BEGIN SIGNATURE-----
MCwCFGsXgW0vI18VHuej4dJM7+W+RiRuAhQER9V2GTfiG+6/Wy6TrIna5MDgIw==
-----END SIGNATURE-----
Name: Simon
Email: simon@linuxha.net
```

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

```
# mkdir /tmp/tmpdir
# cd /tmp/tmpdir
# gunzip -c package.tp2 | tar xvf -
# [[ -f TP2_PACKAGE ]] && gunzip -c TP2_PACKAGE | 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.

## 2.1 The “!..config..!” File Contents

Apart from the files the package actually delivers the only other file present in the archive is a file called “!..config..!” which contains the meta-information regarding the package content. The actual contents of this file are variable, but will typically at least include:

- Package Name
- Package Version
- Short Description
- Directory List
- File List

Optionally it will include a lot of other information, possibly including:

- Trigger Scripts (pre/post/final install/remove, check install)
- License, Readme, Copyright Files
- Architecture
- Operating System
- Dependencies
- Incompatibilities

### 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:

Type	Purpose
<b>Generic</b>	<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 are compatible across 32 and 64 bit platforms and different CPU endian-ness.</p>
<b>OS Specific</b>	<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>
<b>Architecture Specific</b>	<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>
<b>OS and Architecture Specific</b>	<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 and 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 included, 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.

### 3.2.1 Implicit or Explicit Directories

If the required directories for the packaged files are listed in the package they are considered "explicit directories"; whilst if they are not but need to be created to allow the package to deploy files they are considered "implicit".

The difference between the two becomes apparent really when a package is removed – at that time explicit directories are removed (if no other packages include them as explicit directories) whilst implicit directories are left on the file system.

### 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.

```
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** use 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 automate calling of build scripts. In this case generated package name was as follows, since it was created on a 32 bit Linux host:

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

## 4.1 Use of 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 has 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 many architecture and OS variants then the developers must make the effort to generate many packages for each set of architectures and/or OS variants they intend to support.

## 4.2 Use of 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.

## 5 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.

### 5.1 When scripts can be run

The TP2 software suite is able to run scripts when packages are installed and removed, as described in the following table:

LABEL	Purpose
<b>PREINSTALL</b>	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.
<b>POSTINSTALL</b>	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.
<b>FINALINSTALL</b>	Very similar to a "post-install" script; however the return code issued does not stop the package from being considered as installed (although any failures are logged as events in the audit log facility TP2 offers).
<b>PREREMOVE</b>	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.
<b>POSTREMOVE</b>	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.

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/Bash shell compatible scripts using the `"/sbin/sh"` executable. For truly generic packages it is recommended that such scripts only basic shell functionality and should using the following `"#!"` line:

```
#!/usr/bin/env sh
```

## 5.2 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>"FINALINSTALL"</code> , <code>"PREREMOVE"</code> , <code>"POSTREMOVE"</code> and <code>"CHECKINSTALL"</code> .

When the script is run it is done so from the package root directory, so initially when a script is run the current working directory is the top level directory for the namespace the package is being installed into.

If the script is the "check install" script, then two additional environment variable will be made available:

Variable	Purpose
PKGMSGFILE	A file which the calling script can write to allowing it to dynamically alter the list of dependencies that must be installed for this package to be installed.
PKGDEPENDS	A white-space separated list of dependencies defined for this package that must be installed for this install to work.

If the script being run is a "pre-install" or a "post-install" script, then the following are available also:

Variable	Purpose
PKGINSTALL_ACTION	Indicates the type of installation taking place; <b>install</b> - Installation of a package that is not already installed. <b>reinstall</b> - Re-installation of the same version of an already installed package. <b>downgrade</b> - The package is already installed; an older version is replacing it. <b>upgrade</b> - The package is already installed; a newer version is replacing it.
PKGVERSION_PREVIOUS	If the package is already installed [and the action described above is not "install"] this will contain the version number that is currently installed.
PKGS_INSTALLED	A list of packages that are already installed in the namespace prior to the "tp2 install" command being run. All packages are white space separated, and each package given is in the format "pkhname pkgversion".

### 5.3 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.

#### 5.3.1 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)

#### 5.3.2 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.

### 5.3.3 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.

### 5.3.4 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.

### 5.3.5 Final-Install Script

This script is run at the very end of successful package installations (following the post-install and package clean-up). Often it is used to generate "standard output" messages to ensure that information is passed to the installer regarding the package. The return code does not impede the package installation, though a non-zero return code will register in the audit log.

```
#!/usr/bin/env sh
echo "Please run the following to read license:"
echo
echo "tp2 list --namespace $PKGSPACE --show readme $PKGNAME"
echo
```

## 5.4 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>
  <preremove>pkg/preremove</preremove>
  <postremove>pkg/postremove</postremove>
  <preinstall>pkg/preinst</preinstall>
  <postinstall>pkg/postinstall</postinstall>
  <finalinstall>pkg/finalinstall</finalinstall>
  <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.

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.

## 5.5 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
<b>PREINSTALL</b>	May check to see if an existing version of the package is already installed, and if so may stop existing programs.
<b>POSTINSTALL</b>	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.
<b>PREREMOVE</b>	As with a pre-install script this may ensure that any daemons that might be running for the deployed package are stopped.
<b>POSTREMOVE</b>	If a program generated binary programs as part of the post-installation script the post-remove script should be written to remove such files.
<b>FINALINSTALL</b>	Typically to show a message to get the user to do something now the package is definitely installed/upgraded/downgraded as necessary.

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 cannot 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:

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

Future enhancements due in the future for 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.

A simple final installation script might simply show a message, or even indicate details of other packages that might be installed to be work alongside the package just installed.

## 6 Usage Package Verify Scripts

### 6.1 Purpose of Verify Scripts

Although many package management toolsets have the ability to perform verification of package installations (as does TP2), this facility has been extended by allowing packages to optionally define a verification script.

If a package needs to be verified (via a user/admin calling the "tp2 verify" facility), then if a package has a verification script it will be executed as part of this process.

The intention is to allow packagers to include some code which can perform some checks beyond the standard verification steps to ascertain the status of the package. The script specified is run in addition to the standard verification steps, and so has no need to check the following:

- Installed Files - the contents, permissions, owner and group of the files installed as part of the package.
- Directory permissions - the owner, group and permissions of any directories defined as part of the package.

So typically a verification script might ensure that the steps performed by any post-installation script have been successful. That is any start-up scripts are in place, kernel configuration options are suitable, etc.

### 6.2 Script Environment

The verification script can make use of the following environment variables if necessary:

Variable	Purpose
PKGROOT	The name space root directory, such as "/opt" or even "/home/sedwards".
PKGSPACE	The name of the current namespace the package is being installed to.
PKGNAME	The name of the package being installed.

The verification script should write any problems to standard output. The return code from the script indicates verification status:

- 0 – Package verification script was successful.
- 1 – Package verification script issued warnings.
- 2 – Package verification script failed (package verification failure)

## 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
<b>PKGROOT</b>	The name space root directory, such as “/home/sedwards”.
<b>PKGSPACE</b>	The name of the current namespace the package is being installed to.
<b>PKGNAME</b>	The name of the package being installed.
<b>PKGVERSION</b>	The version number of the package being installed.
<b>PKGACTION</b>	The action that the script has been called for – will be either “PREINSTALL”, “POSTINSTALL”, “PREREMOVE”, “POSTREMOVE” and “CHECKINSTALL”.
<b>PKGMSGFILE</b>	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.
<b>PKGDEPENDS</b>	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
<b>=DEPENDS</b> <b>[dep1[,minver=N.N.N,maxver=N.N.N] ...]</b>	The complete list of original dependencies is replaced by this list. Must only occur once in the file.
<b>&gt;DEPENDS</b> <b>[dep1[,minver=N.N.N,maxver=N.N.N] ...]</b>	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.
<b>-DEPENDS [dep1 ...]</b>	Removes the specified packages from the list of dependencies (if they are currently included). Can occur multiple times.
<b>-DEPENDS *</b>	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 Understanding 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	<b>Version 1 is the latest version.</b>
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 3<sup>rd</sup> 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 cannot 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 "TP2PATH" 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 "TP2PATH" 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 cannot 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 Text File Support

A package definition can optionally specify one or more text files that are part of the package – this can be useful to capture certain information. The text file labels currently supported are:

Label	Purpose
<b>description</b>	A long description of the package.
<b>readme</b>	A file containing information regarding the package installed.
<b>license</b>	The license that the software package has been released under.

The following lines in the package configuration file are used to indicate the files to register

```
<readme_file>pkg/readme</readme_file>
<description_file>pkg/description</description_file>
<copyright_file>pkg/copyright</copyright_file>
```

Obviously these files must be present otherwise the package build will fail. There are no limits on the size of the file that can be registered using this process. However the file should be a text file that should be possible to display to a standard terminal.

When text files are registered with a detail the user can issue a command at any time to view the contents of that file, for example:

```
$ tp2 list --namespace myns --show readme my_package
```



## 11 Working with Package Bundles

### 11.1 What are Bundles?

A bundle might be considered as a collection of packages that can be conveniently installed and managed. A bundle is different from a normal package:

- It does not include "<file>" or "<directory>" elements.
- It includes a "<bundle>" element to describe the packages that make up the bundle.

So a bundle works in a similar way to Debian "meta-packages" - that is a bundle contains no software itself, only references to other software. Compare this to AIX "install" package bundles or HP-UX's SD-UX filesets which physically contain all relevant software.

```
<bundle>
  <pkg minver="1.0.0">pack2</pkg>
  <pkg minver="1.0.0">newfred</pkg>
  <pkg minver="1.1.0">tp2</pkg>
</bundle>
<incompatibles>
  <pkg>oldfred</pkg>
</incompatibles>
```

Notice that in the above fragment that a bundle can still include other attributes that a normal package might include - an "incompatibles" section in this instance.

### 11.2 Building a Bundle

Building a bundle is identical to building a package, for example:

```
$ tp2pkg --config pkg/conf --verbose
Log : Well-formed XML in "pkg/conf" - continuing.
Log : Validated XML in "pkg/conf" - continuing.
Log : Packaged will be spooled to: /tmp/test-bundle+2.2.0.tp2
Log : Package generated successfully.
```

Easy!

### 11.3 Installation and use of Bundles

A bundle can be installed with the usual install command, for example:

```
$ tp2 install --namespace testns2 --pkg test-bundle
```

When a user performs a "tp2 list" a bundle can be seen by a line which is followed by indented lines beginning with a "+", for example:

```
$ tp2 list --namespace testns2
Package          Version          State           Installed
=====
act              0.2.5           Installed      21/11/2012
filer           1.0.0           Installed      11/12/2012
fred            1.1.0           Installed      30/10/2009
prospect_production 2012.01.0902    Installed      30/01/2012
test            1.0.0           Installed      08/03/2012
test-bundle     2.2.0           Installed      05/11/2009
+newfred        2.1.0           Installed
+pack2          1.2.0           Installed
+tp2            1.1.6           Installed
testpkg         1.0.0           Installed      11/12/2012
```

Consider the following:

```
$ tp2 remove --pkg pack2 --namespace testns2
```

This will be allowed since bundles are not packages - they might have dependencies, but nothing stops the user/admin actually removing part of the bundle. Following this action, consider the "tp2 list" output shown:

```
$ tp2 list --namespace testns2
Package          Version          State           Installed
=====
act              0.2.5           Installed      21/11/2012
filer           1.0.0           Installed      11/12/2012
fred            1.1.0           Installed      30/10/2009
prospect_production 2012.01.0902    Installed      30/01/2012
test            1.0.0           Installed      08/03/2012
test-bundle     2.2.0           Partial        05/11/2009
+newfred        2.1.0           Installed
+tp2            1.1.6           Installed
testpkg         1.0.0           Installed      11/12/2012
```

Notice now that the state for the bundle is now "partial". Listing the details for that bundle will indicate why it is partial:

```
$ tp2 list --namespace testns2 --detail test-bundle
Name           : test-bundle
Version        : 2.2.0
State          : Partial
Bundled        : newfred (2.1.0) [1.0.0 -> 9999999999.999999.999999]
                : pack2 MISSING
                : tp2 (1.1.6) [1.1.0 -> 9999999999.999999.999999]
Incompatibles  : oldfred [0.0.1 -> 9999999999.999999.999999]
Scripts        : Postinstall
Text Files     : Copyright,Description,License,Readme
Altrepos       : WWW:myhost/tp2packages
Description    : An example bundle
Install Date   : 05/11/2009
```

To "fix" the bundle there are two approaches:

1. Perform a re-installation of the bundle
2. Install the missing package

## 12 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.

### 12.1 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>
  <bfile perms="644" owner="root" group="sys"
    configfile="1">cfg/pkg.conf</file>
</files>
```

### 12.2 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>
  <bfile perms="644" owner="root" group="sys"
    volatile="1">var/example_log</file>
</files>
```