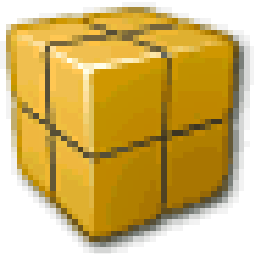
PackageShell



INTEGRATED SOFTWARE INSTALLATION SERVICE

Updated: July 9th, 2019
Version: 4.3 Build 431

Contents

1 Introduction

In environments with a large number of clients, software distribution is typically handled by products like Microsoft System Management Server (SMS) or IBM Tivoli. PKGSHELL adds to and enhances the functionality provided by these products in several aspects:

1.1 Better end user experience

- The user is guided through the installation once a package reaches a client. The user gets a simple, consistent interface to start and monitor the installation.
- Logoff of the end user is controlled and enforced if needed. This is so the user does not interfere with the execution (fewer errors) but also that the installation does not interfere with the user's work (fewer complaints).
- The installation progress is made visible to the end user. Unlike with other agents, the currently visible desktop is used to execute installations.

1.2 Greater transparency for the administrator

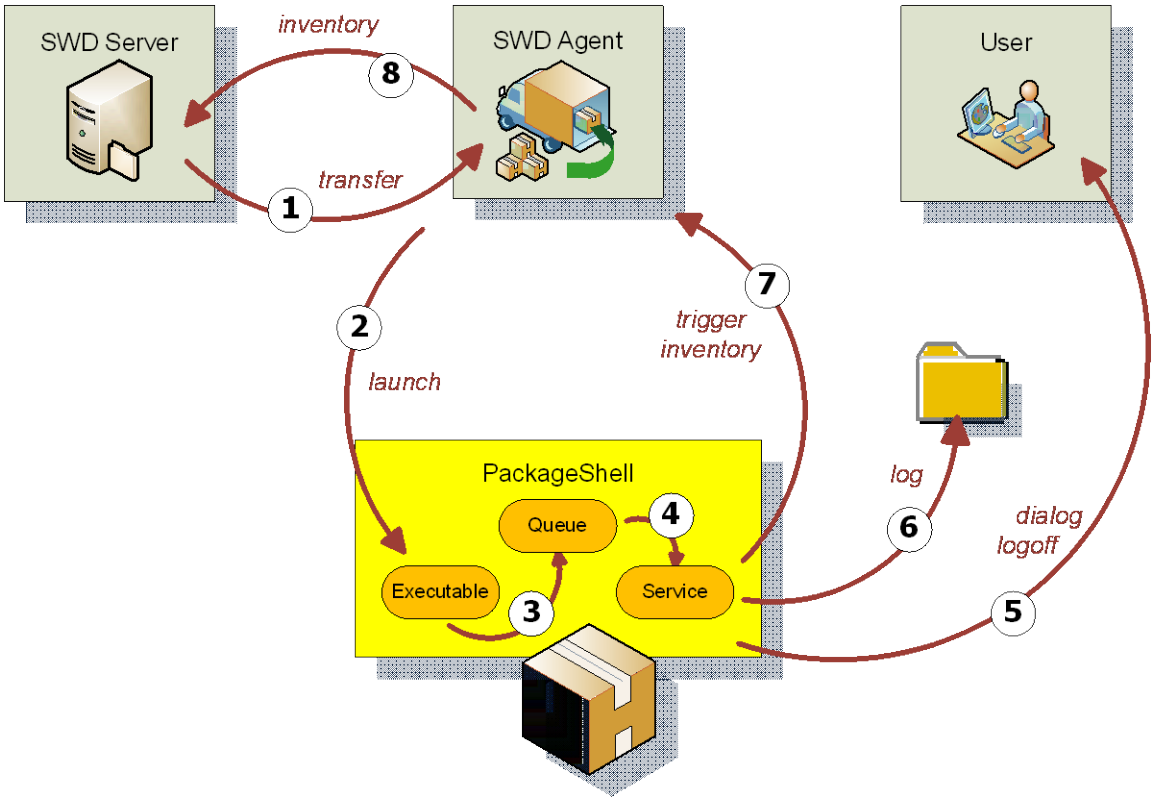
- Package status and activity history are reported by customizing the hardware inventory of the clients.
- Detailed milestones are reported to indicate installation progress on each client.
- Error conditions like power-offs and network failures during the installation are intelligently detected and reported.

1.3 Enhanced productivity for the package author

- Robust support to control reboots as part of an installation. If a reboot is performed, the installation is considered successful only after the client is back online. An installation can be made of several parts with unlimited optional reboots in-between.
- Convenient support to define tests before, during and after an installation. The powerful yet easy built-in test language can eliminate the need for hard to maintain external test scripts.
- Tight integration with Microsoft Windows Installer (MSI) makes use of MIF-file analysis, log file control and standard MSI parameters.

2 Overview

2.1 Flow of execution



Steps 1 and 2 above are the normal process flow of software distribution. After the PKGSHELL.EXE is run in step 2, the package is already completed from the point of view of the software distribution system; SMS will effectively report a status of „Completed" for the advertisement.

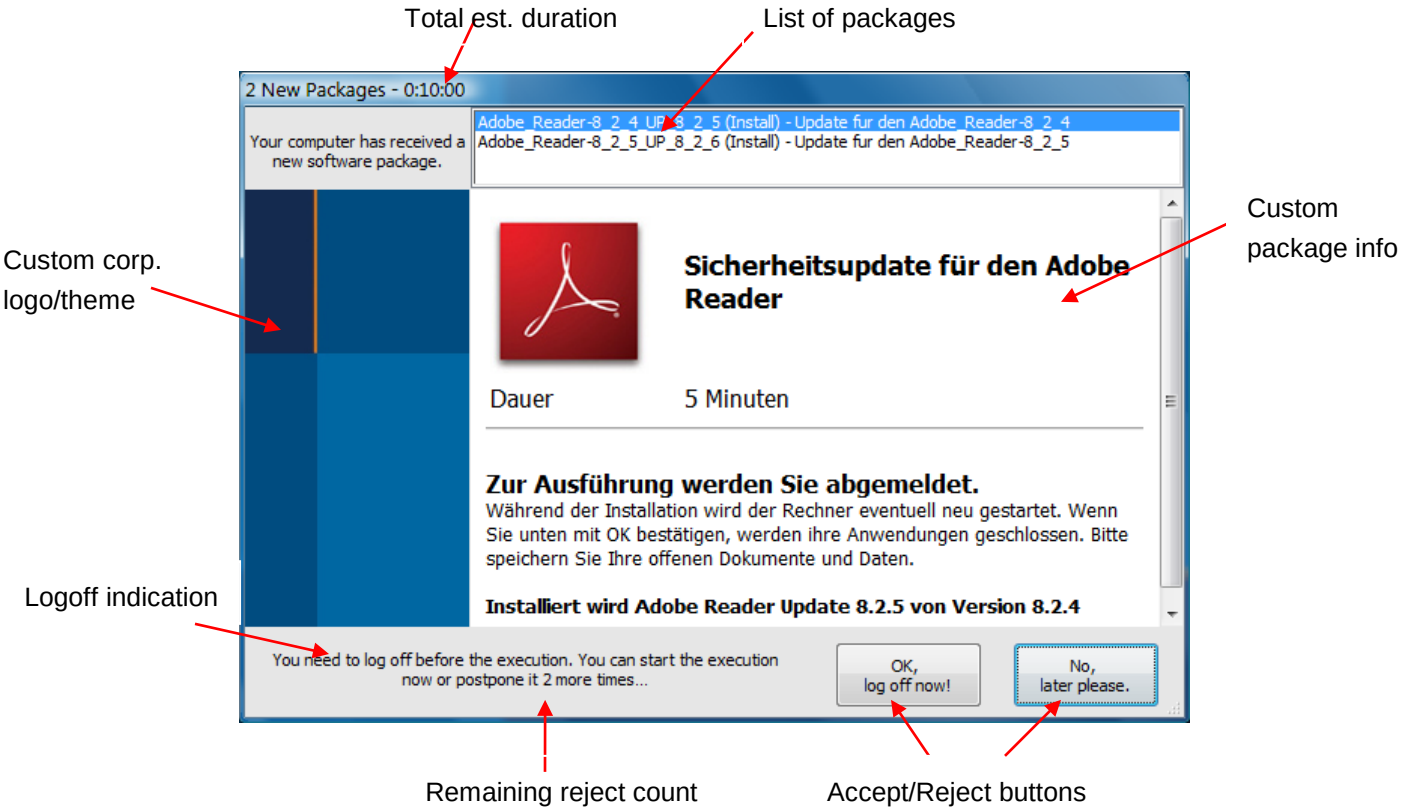
In step 3, the executable will place the package definition into the local queue. Technically this means that the Ini-File that makes up the package definition is parsed and stored in the local registry, the „queue".

In step 4, the service processes the package that is stored in the queue. As needed, the user will be informed and logged off (5) before execution. All activities are logged to standard locations (6). When the execution is completed, the inventory is triggered to update the packages status back to the SWD system (7).

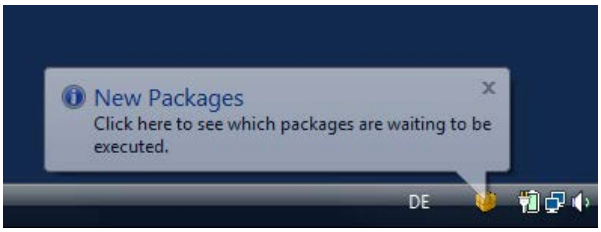
Note that because PKGSHELL performs the execution steps independently of the SWD system, it does not make any difference if the PKGSHELL.EXE was initially run by the SWD system or if it was started manually.

2.2 User Interface

When the PKGShell service starts while a user is logged on, PKGShell will inform him about the pending installation and ask him whether the execution should start now or whether he wants to postpone it to a later time. The dialog is dynamically localized into the language of the currently logged on user.



If he accepts to proceed immediately, then PKGShell will initiate a logoff. This actually helps because the user may not know how to perform a logoff. If the installation is postponed, the dialog is minimized into a tray notification icon. The user can now continue with his work. The dialog box will reappear after 5 hours.



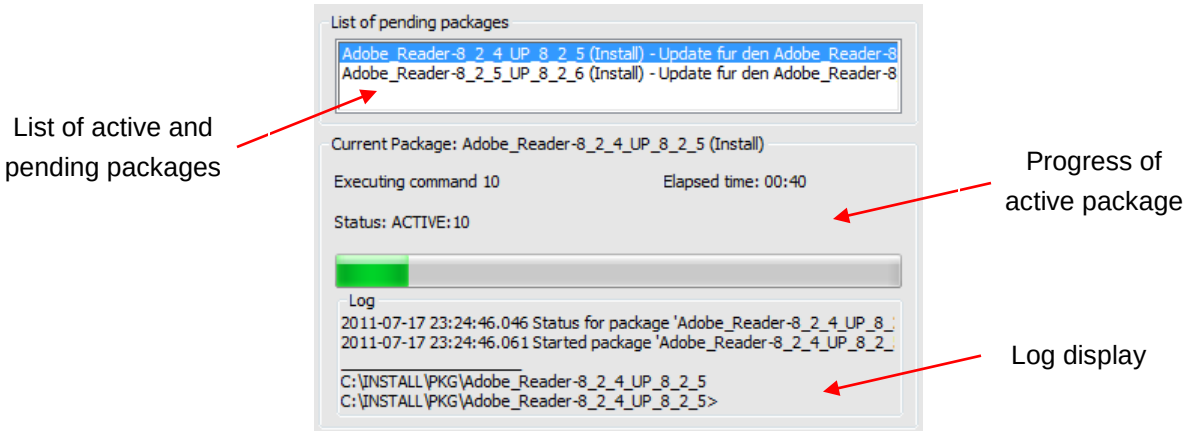
If another package arrives for execution, it is queued. If the user has rejected the execution of the previous package, the tray icon will display a balloon tip to indicate the arrival of a new package.

If the computer is put into presentation mode (supported on Windows Vista and above) then the first dialog never pops up. The balloon tip will also be suppressed until the

machine comes out of presentation mode. This is to support minimal interruption of the user in critical situations like presentations or other full-screen applications.

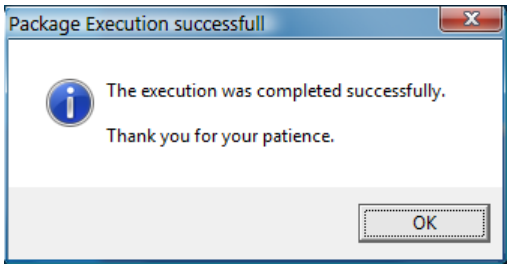
When the execution finally starts, the background wallpaper of the logon desktop is changed to a custom installation background. The „Control-Alt-Delete" prompt and the user logon screen are hidden so the user cannot log on until the execution is finished.

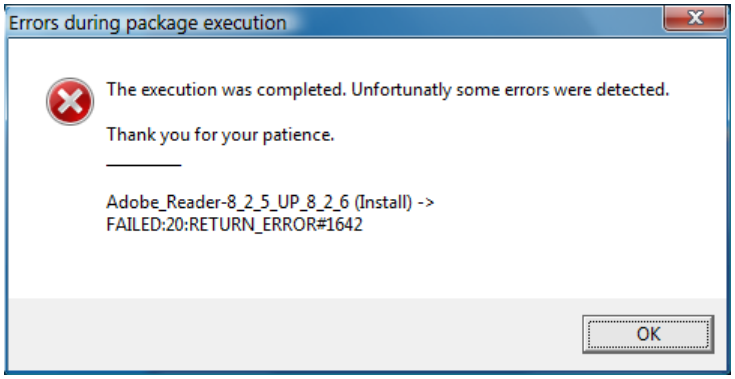
During the execution, a progress dialog indicates what packages are queued and pending to be installed, which command inside the package is just being executed and what the last lines of the log file currently are. A progress bar is optionally displayed if an estimated execution time is specified in the package.



After all packages in the queue are processed, the background is restored to the default wallpaper and the logon dialog box is unhidden.

If a user has previously accepted the execution, a final dialog box informs the user whether all packages were executed successfully:





The error details are shown in the case that at least one package was completed with an error.

2.3 Four modes of execution: Dialog/Logoff/Direct

Depending upon which parameters are used for a package, the execution is performed in a different environment. The relevant parameters which control this behavior are „Direct", „Logoff" and „Dialog". If a parameter is not specified, it defaults to 0. There are four allowed possibilities to combine these parameters into modes of execution:

	Direct=	Logoff=	Dialog=
Silent Mode (Default)	0	0	0
Dialog Mode	0	0	1
Logoff Mode	0	1	*(ignored)*
Direct Mode	1	*(ignored)*	*(ignored)*

Following is a detailed explanation of the four modes.

2.3.1 Logoff Mode (Logoff=1)

If „Logoff=1" is specified, PKGShell makes sure that the user is not logged on during the installation.

The installation will start immediately if no user is active when the package is queued. If a user is active, he will see a dialog to inform him of the pending installation and he will eventually be logged off. The user is prevented from logging on during the course of the installation.

The execution is done in the context of the local SYSTEM account.

This mode should be considered the preferred mode because, by experience, this mode guarantees the highest success rate for packages. The user has the advantage that he sees what his machine is busy doing and clearly knows when this is done.

At first you might be scared of this radical approach because you are disrupting the user every time he gets new software. Truth is the software delivery business is about disrupting the user. If you keep users logged on, his machine usually will still be heavily impacted by the installation activities. So if he is unhappy anyhow it is more honest to log him off for the time needed. The time lost by logging off and on again is in most cases well invested considering the higher success rates of the installations.

2.3.2 Dialog Mode (Dialog=1)

This mode allows for a visible execution with the user still logged on.

The installation will start immediately if no user is logged on when the package is queued. If a user is active, he will see a dialog to inform him of the pending installation but he will not be logged off after he starts the execution.

As with „Logoff=1” the execution is run under the local SYSTEM account.

This mode has the restriction that reboots cannot be performed since the user may still be logged on.

2.3.3 Silent Mode (Default)

This mode is the default if you do not specify anything else in the package control file, meaning all of Dialog=0, Logoff=0, Direct=0.

The package is executed immediately after queuing, regardless of whether the user is logged on or not. The user will not see any information before execution is started and he will not see the PKGShell progress dialog.

However, if the package itself displays any UI elements, these will appear on the active desktop. If the package requires some form of interaction, this is made possible by always using the active desktop – which is either the user’s desktop if he is logged on, or the logon desktop if nobody is logged on.

As with „Logoff=1” the execution is run under the local SYSTEM account.

2.3.4 Direct Mode (Direct=1)

The use of this parameter completely overrides the parameters ‘Logoff’ and ‘Dialog’. The resulting behavior is actually similar to „Logoff=0”, but it goes a bit further:

This very different mode is called „**Direct**” because no queue is used. It differs from the standard procedure in that it leaves out steps 3, 4 and 5 of the above process diagram:

- Packages are executed immediately without the use of a local queue.
- The installation is run in the context of the calling user.
- The PKGShell service is **not** installed or started and the PKGShell.exe is not copied locally.
- The user is not informed about the pending installation and he does not get a chance to postpone the execution.

There are some severe restrictions for „Direct=1” that you should be aware of:

- Reboots are not supported as part of the installation. All reboot requests will be ignored. If a reboot is nevertheless performed, the package will not be completed and no final status will be reported.
- Nesting of packages using the command „PKG:" is not supported.
- If an installation „hangs" the user cannot respond to any UI elements, since these will not be visible.

These restrictions limit the use of this parameter to only very simple packages or scripts. It may still be preferable to run packages using „Direct=1" instead of not using PKGShell at all – because you get the benefits of standardized reporting and logging.

If you just want to execute packages silently without logging off the user, you should prefer „Silent" mode (parameters **Logoff=0** and **Dialog=0**) which will, similarly, run the package without interfering with the user. „**Silent**" is better because it runs consistently under the same user account (local system) as queued packages. It further does not have any of the restrictions in functionality that Direct=1 has.

2.3.5 Summary

The short story is:

- You should generally be using „Logoff" mode because it's the safest and most flexible.
- Use „Dialog" if you are sure a package does not need reboots and also cannot be disturbed by anything the user is doing while it is executing.
- Use „Silent" mode if the package does not install anything and does nothing that could disrupt the user.
- Use „Direct" mode for little maintenance scripts and the like that have no dependency on any other installations that are going on.

Each of the above modes has its specific restrictions which are summarized in this table:

	Support for Reboots	Consent and Progress Dialogs	Order of Packages	Nested Packages
Silent Mode	No	No	Overtakes Logoff	Yes
Dialog Mode	No	Yes	Preserved	Yes
Logoff Mode	Yes	Yes	Preserved	Yes
Direct Mode	No	No	Immediate	No

2.4 Guidelines

The following best-practices for software distribution are not required but were influential during the design of PKGShell:

- Installations should generally be performed while the end user is logged off. This ensures consistent results and eases testing for the packaging team. The end user is also better off because he is clearly notified of installation activities and there is less he can do wrong.
- Installations should run in the context of the local system account.
- Installations should use MSI (Windows Installer) technology whenever possible. MSI technology gives installations a uniform interface, detailed logging, rollback in case of error and uninstall functionality.
- Installations should not write to the HKEY_CURRENT_USER part of the registry. This is a natural requirement for installations in order to function on a computer used by more than one end user.
- The client workstations are expected to be „locked down" – the end user should usually not have administrative rights.

2.5 Supported operating systems and dependencies

The tested and supported operating systems that PKGShell runs on are:

- Windows 2000 Professional and Server Editions
- Windows XP / Server 2003
- Windows Vista / Server 2008
- Windows 7 / Server 2008 R2
- Windows 8 / Server 2012
- Windows 8.1 / Server 2012 R2
- Windows 10 / Server 2016

PackageShell is developed in native C++ and has no dependency on any run-time-environments or other DLLs. The whole product consists of a single executable (about 200 KB for the 32bit version) with optional language DLLs.

No development frameworks like .Net Framework, MFC or other Runtime Libraries are used - because PackageShell itself is commonly used to install or reinstall such system components.

The PKGSHELL.EXE needs to be started with **administrative rights** on the client. This is required even if the execution of the package itself would not need administrative privileges.

3 Usage

3.1 Installation

PKGShell should be installed on the local computer before any packages are executed. This ensures that any customizations (like log file and registry paths) are in effect before the machine starts building its history of packages.

The PKGShell distribution comes with 2 executables, one for 32bit Windows and one for 64bit Windows. On 64bit Windows ***both*** executables should be installed. Technically the installation is performed using the parameter „-i" at least once on the local machine:

```
pkgshell.exe /i
```

This will install the service EXE along with the language DLL files in the correct System32 windows directory.

You can use the provided `install.cmd` batch file from the distribution to perform the installation with optional customizations in 3 quick steps:

1. **Customize Config File (Optional)**:
   Edit **PKGShell.cfg** and add your company's name as `RegRoot` parameter.
   Add your provided license.

2. **Customize Exe Name (Optional)**:
   Edit **PKGShell.sms** and edit the line `exename=`***`newname`***
   Execute `ren pkgshell.*` ***`newname`***`.*`

3. **Perform Installation**:
   Execute `install.cmd`

Please see section 6.2.26, page 56, for more information on customizing PKGShell.

It is recommended to implement the call to install.cmd in a package that is included both in your master image as well as mass deployed to all existing machines in your organization.

☞ PKGShell has an „On-Demand" installation behavior if you place the PKGShell files inside the package you want to execute. Although it works, it is not recommended to rely on this method on 64bit OS installations since this will only install the 32bit version.

3.2 Running PKGShell.exe

A package is always started by running the executable PKGShell.exe. The EXE should already reside in the default windows path of `%WINDIR%\System32` if installed correctly. By default, the package to be processed is identified by changing the ***current directory*** before calling PKGShell.

The executable PKGShell.exe can additionally be placed in each package's source directory to serve as a „launcher" for the package. In that case, the Exe in the package directory will queue the package, but the locally installed Exe will actually run the package.

The calling syntax is

```
pkgshell.exe [program] [-f path] [option]… [switch]…
```

«program» is an optional parameter that defaults to «install». PKGShell.exe typically does not need more command line parameters because all necessary parameters are read from the package definition file. The package definition file is searched for in

1. the optional path specified by -f
2. the current directory
3. the directory that PKGShell.exe was run from

PKGShell.exe will stop further processing if it cannot successfully locate and read a package definition file.

«options» are package parameters that can be used to override program parameters from the package definition file. If your package definition file has „Logoff=1" but you would temporarily like to run it with „Logoff=0" without changing the package file, you could say:

```
pkgshell.exe /logoff=0
```

Any of the package program parameters can be specified on the command line. See the list of package program parameters in section 8.1.2 on page 65.

«switches» are directives that tell PKGShell to do something different than running a package. E.g. to delete all queued packages from the local queue:

```
pkgshell.exe /x
```

See section 8.2 on page 68 for a complete reference of available command line switches.

3.2.1 Queuing many packages at once

If you need to execute many packages at once, you need to call the EXE for each package that you want to queue. By default this will result in the info dialog box to re-appear for each package. Because this is annoying for the user and slows things down, there are two options to prevent this:

1. For quicker operation you can use the parameter „-Q" (Queue-Only) which does not restart the service after adding a package to the queue.
2. You can also activate the presentation mode to prevent the dialog box to open and close unnecessarily.

To kick of the installation once all packages are queued, you can start the service so the consent / info dialog box will appear.

If you wish to start the execution without getting the user's consent, use parameter „-K" (Kickstart). Be careful that the user might lose any unsaved documents.

3.3 Return codes

The PKGShell.exe will generally return 0 (zero) upon successful execution. If the return code (a.k.a. `Errorlevel`) is not 0, then it is an error code that was returned by the windows API function that led to the error.

There are some few return codes that are defined and returned directly by PKGShell:

Number	*Status*	*Explanation*
800	**ERROR_PKG_SKIP**	Test:Required failed
801	**ERROR_PKG_PREQUEUE**	Test:PreQueue failed
802	**ERROR_PKG_PRERUN**	Test:PreRun failed
803	**ERROR_PKG_CANCELED**	Package Status is CANCELED
804	**ERROR_PKG_ABORTED**	Package Status is ABORTED
805	**ERROR_PKG_FAILED**	Package Status is FAILED
806	**ERROR_PKG_SUCCESS**	Test:Success failed

4 Feature Details

4.1 States and Milestones

Most of the time PKGShell will be waiting for external activity; E.g. waiting for the user to respond, waiting for completion of the executed command or waiting for a reboot to complete. It is important to know in what state the package currently is for troubleshooting purposes. Reporting only one state when the package has completed would not be enough – that's why PKGShell defines many intermediate states.

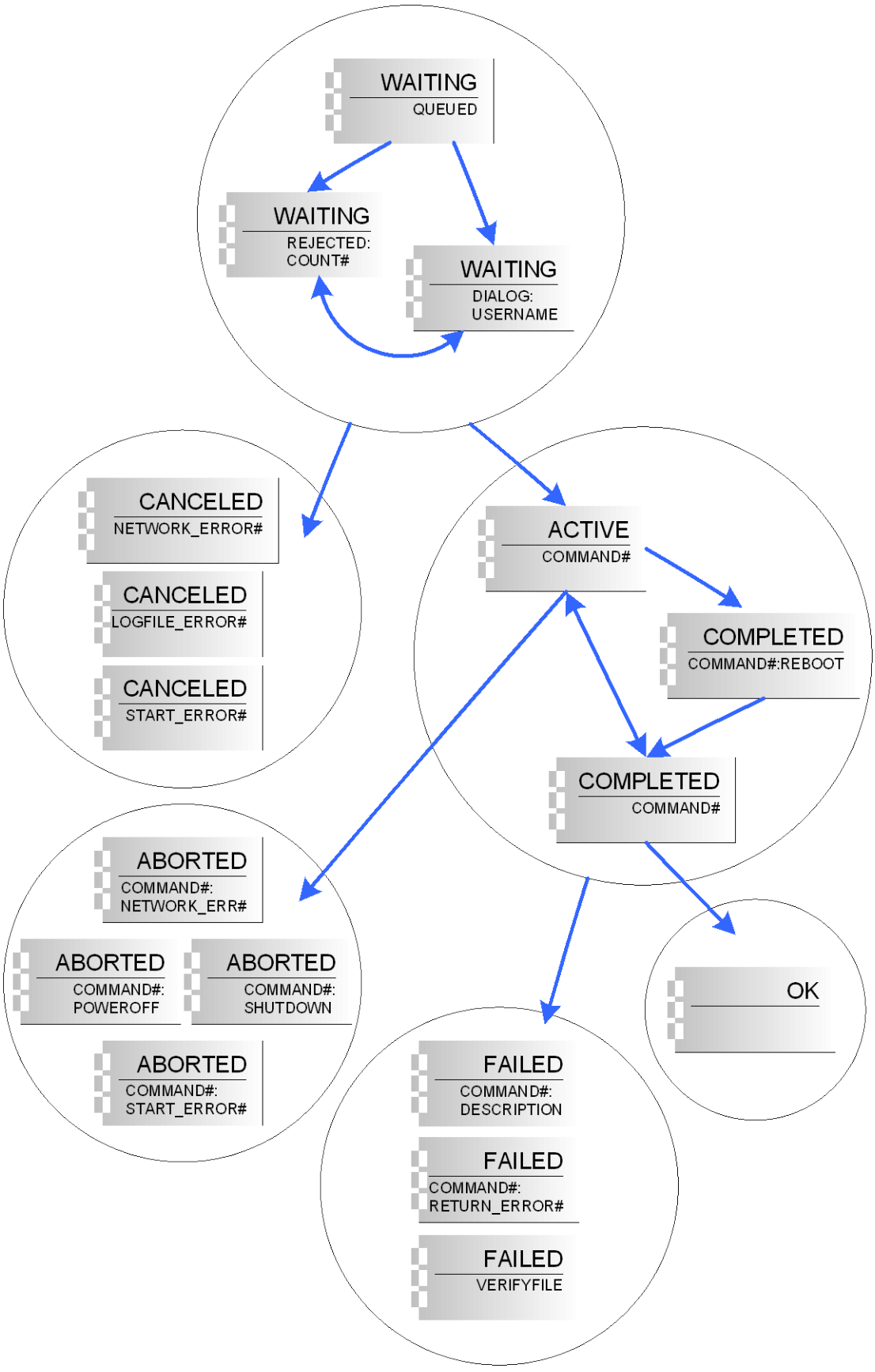
The diagram on the following page depicts which states PKGShell can go through. Notice the difference between final states (no arrows leaving) and transitional states (non-final). A package is completed when it has reached a final state.

The only positive final state is 'OK'. Negative final states are 'CANCELED', 'ABORTED' and 'FAILED'. This differentiation gives more information about the time, source and severity of the error:

CANCELED means that the error occurred so early that no command was ever started. This is the least critical situation, because the client is still untouched and thus the package could be simply restarted.

ABORTED means that some environmental error occurred, that is beyond control of the executed command. This could be a power failure or a network failure. This is the most critical because it potentially leaves the client in an intermediate state that cannot easily be recovered.

FAILED means that an error occurred within the executed command and that the error was reported by the command itself.


WAITING
QUEUED
WAITING
REJECTED:
COUNT#
WAITING
DIALOG:
USERNAME
CANCELED
NETWORK_ERROR#
CANCELED
LOGFILE_ERROR#
CANCELED
START_ERROR#
ACTIVE
COMMAND#
COMPLETED
COMMAND#:REBOOT
COMPLETED
COMMAND#
ABORTED
COMMAND#:
NETWORK_ERR#
ABORTED
COMMAND#:
POWEROFF
ABORTED
COMMAND#:
SHUTDOWN
ABORTED
COMMAND#:
START_ERROR#
OK
FAILED
COMMAND#:
DESCRIPTION
FAILED
COMMAND#:
RETURN_ERROR#
FAILED
VERIFYFILE

4.2 Logging

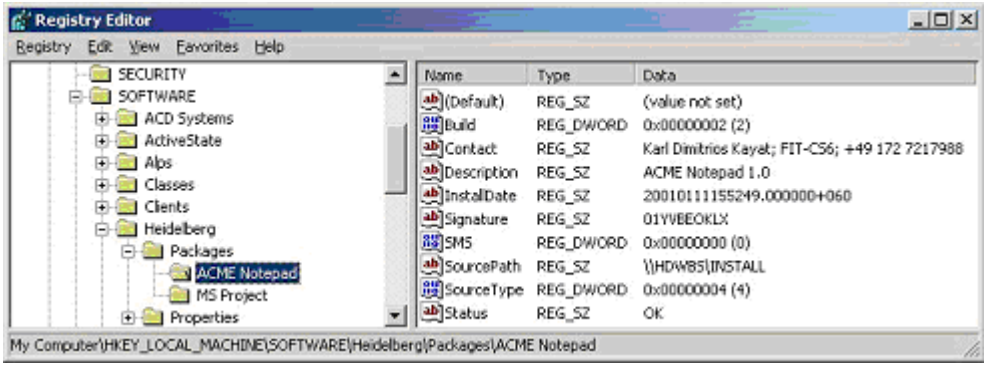
For each state or milestone that is reached, the current state is

4. written to the **registry** under HKEY_LOCAL_MACHINE\SOFTWARE**Company**\Packages***Package***\Status
5. appended to the central PKGShell log file (%windir%\Logs**pkgshell.**log)
6. logged to the NT application **event log**. The NT application event log also contains detailed information about all warnings and errors that PKGShell encounters.

The output of all commands of a package is written to a separate **package log** file named %windir%\Logs***PackageName***.log.

4.3 Reporting

For each package that the client receives, a registry key is created under HKLM\Software\Company\Packages path:



Generally all parameters that were given through the package definition file are replicated here. PKGShell additionally records some dynamic properties for the package:

InstallDate	Date and time of the package's last status change in DMTF format	20040229
Duration	Duration of the execution in seconds	90
SourcePath	The directory file path to the installation sources. This provides statistics over what servers were used.	C:\Temp or \\SRV\VOL1\…
SourcePathOrigin	The original source path if the CopyLocal parameter was used	\\SRV\VOL1\…
SourceType	Type of the source path drive:	Usually 4

	Removable (2), Fixed (3), Remote (4), CDROM (5)	
Signature	Branding hash string	

The signature field uses an algorithm to produce a hash so the authenticity of the installation status can be verified. It contains a hash value of the fields Build, Name, Computername, Status, SourcePath and InstallDate.

4.4 Footprint: PKGShell EXE and service

The same PKGShell.exe binary has two operation modes:

1. When executed, it adds a package to the local queue, copies itself to the local System32 directory, installs and starts the PKGShell service and finally reports the queue operation status to SMS by placing a status MIF in the Temp directory.
2. When started as a service, it interacts with the logged on user and processes the locally queued packages.

The windows service named '`PKGShell`' is configured to start at boot time. It adds the following entries to the system registry:

`HKLM\Software\Company\PKGShell`
: Location of the package queue

`HKLM\System\CurrentControlSet\Services\PKGShell`
: Standard entry for the system service

`HKLM\System\CurrentControlSet\Services\Eventlog\Application\PKGShell`
: Event source entry for displaying PKGShell event logs

Packages in the local queue are generally processed in a first-in-first-out order. A package is removed from the queue after it is completed. When no packages are left in the queue, the PKGShell service terminates.

On 64bit systems the following registry links are automatically created to ease the interoperability of 32bit and 64bit executables:

`HKLM\Software\Wow6432Node\Company\PKGShell`
→ `HKLM\Software\Company\PKGShell`
`HKLM\Software\Wow6432Node\Company\Packages`
→ `HKLM\Software\Company\Packages`

All changes are thus visible in both 32bit and 64bit locations.

4.5 Environment of package commands

When PKGShell executes a command, it does so in an environment that may differ from what some installations expect. Below is an outline of these peculiarities to be aware of when developing and troubleshooting a package.

- The process is running as the local '**System**' user. This user has full access to all resources on the client – even more access than the local administrator account.

- Because the process is running as 'System' it may not be able to transparently connect to servers and use remote paths. If a command needs access to a network location other than the package source directory, it has to establish a connection explicitly with the credentials of another user (**`net use /user:account password`**).
  Since Windows XP, the System service uses the „computer object account" in domain environments. You could choose to give this account sufficient permission in order to easily access network shares.

- During execution, a drive is temporarily connected to the share that contains the package sources. The **current directory** is changed to the package source directory on that drive. This enables the use of relative paths within the package commands.

- The standard output and standard error handles of the process are both **redirected** to the package's log file. If you are running a console program as a command, everything that is printed to the console will be written to the log file; a GUI mode program can explicitly write to the standard handles too. All output is converted to **Unicode** format on the fly by PKGShell to better support internationalization.

- Windows NT/2000 has more than one logical desktop, most notably the 'Winlogon' desktop that is visible when no user is logged on and the 'Default' desktop which is visible when a user is logged on. PKGShell always starts the command in the context of the **currently visible desktop** to ensure that the user can see the installation progress.

4.6 Closing applications before execution

When the user is not logged off (Dialog=1 and Logoff=0) certain active applications that the user has started may interfere with package installations. It is a common request to terminate these applications before the execution of the package starts.

In that case the parameter `Close=` comes to help:

```
Close = one.exe[:another.exe]…
```

It takes a name of an executable or a list of executables separated with colons (:). If an active window running in the user's context is found active and belongs to one of the given executables, the user will be notified he needs to close the windows by a "Close Applications" dialog.

E.g. to close Microsoft Word, Outlook and any Notepad-like applications you could say:

```
Close = winword.exe:outlook.exe:notepad*
```

The applications are automatically closed if the user confirms the "Close Applications" dialog by pressing "Yes". In case it is preferable that the user closes the application himself, you may use the parameter "`ManualClose=`". With ManualClose=1 defined, the user is only asked to close the applications and has to confirm the dialog pressing "OK".

The execution of the package will only proceed when no more active windows are found. If a user attempts to restart the application after the execution has started, it will be automatically closed without further notice. This is enforced for the entire duration of the package.

- ☞ You should still consider using Logoff=1 as the safest alternative to prevent problems with active applications.

4.7 Waiting for child processes

If the command that is started by PKGShell in turn starts other processes, then PKGShell will wait for all sub-processes to terminate, even if the first child process terminates before the grandchild processes. This is done by inheriting an object handle to all chid processes. PKGShell waits for that handle to be closed – which is what will happen automatically when all processes that inherited the handle are terminated.

This behavior is the default and makes sense for most cases. If you want to start a command and do not want to wait for it to terminate, you need to use the command parameter `.NoWait=1`.

4.8 Integration of reboots

PKGShell supports reboots as an optional but integral part of a package. Whenever a package requires a reboot, it will only be considered successful if the client is up and running again after the reboot.

Because PKGShell is a service, it automatically regains control after the client is restarted. After the restart, PKGShell hides the logon dialog box until all packages have finished executing.

The test for a successful reboot is an important aspect of an installation that, if neglected, may leave the client in an undetected state of failure after the reboot.

For more details on reboots for package authors, see section 5.6 on page 32.

4.9 Integration with MSI

PKGShell integrates with Windows Installer in the following aspects:

1. If a command contains a call to MSIEXEC.EXE PKGShell will automatically modify the Command string to include default MSI parameters suitable for PKGShell.
2. PKGShell correctly recognizes MSI error return codes and reacts to them appropriately.

The MSIEXEC parameters that PKGShell defaults to are:

`/L*+! %LOGFILE%`	(detailed logging to the package log file)
`/M %MIF%`	(status mif file to be parsed by PKGShell)
`/QB-`	(quiet mode)
`REBOOT=R`	(really suppress reboots)
`ARPCONTACT=`	(value taken from package file's Contact parameter)
`ARPNOREMOVE=1`	(disable deinstall through control panel)

For additional information about these settings, search the documentation of the Windows Installer on MSIEXEC command line parameters.

Note that the addition of the parameters is optional. If you have specified either /L, /M, /Q, REBOOT or ARP… with a different value, the PKGShell defaults will not be used for the respective parameter.

☞ To enable de-installations through the Software control panel, you have to override the PKGShell parameter by specifying ARPNOREMOVE= " "

Also beware that if the specified command line does not contain MSIEXEC directly but rather calls MSIEXEC indirectly, then the MSI parameters will not be set by PKGShell. If you are the author of such an 'MSI launcher', it may make sense to pass the above parameters yourself.

The intelligent return codes of the MSIEXEC.EXE are processed as command results. If the return codes indicate that a reboot is required, PKGShell will perform this reboot (see below).

If the command line containing MSIEXEC is a deinstallation using the parameter /X, PKGShell will ignore the return code 1605 (ERROR_UNKNOWN_PRODUCT) if it is returned. It is not considered an error condition when a deinstallation fails because the software was not installed.

4.10 Deinstalling packages

A package program can contain a software deinstallation. If a software package is successfully uninstalled from the client it makes more sense to remove the package key from the registry instead of reporting a status of 'OK' for the package.

To do so, use the Parameter `Uninstall=1` in the program section.

PKGShell will also automatically treat a program as a deinstallation if the program (section) name contains one of the following substrings:

```
deins
delet
remov
unins
entfern
```

These substrings are not case sensitive. If a program with such a name is executed and the final status 'OK' is reached, PKGShell will remove the entire package key from the registry so that it is no longer reported.

If you do not want the automatic behavior, you can always use the `Uninstall=0/1` parameter to explicitly set the type of installation.

4.11 Using a Maintenance Window

Especially when running unattended installations in a server environment, you may prefer to only allow installations to occur in a defined scheduled maintenance window.

You can instruct PKGShell to delay the execution until the next maintenance window time frame is reached by specifying the parameter „Schedule=1" either in the package definition file or on by using the command line parameter „/Schedule=1".

If a package is queued with Schedule=1, the user will not receive a dialog box with the option of starting the installation. The installation will automatically start when the maintenance time is reached and forcibly log off the user. Note that only a user logged on to the console („glass") session is logged off. Terminal Server sessions are not logged off.

The „MaintenanceWindow" parameter is read from the registry value under „HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PKGShell\Parameters". The syntax used is:

Day of Week – Start Time [– Duration]

The duration is optional and defaults to 1 hour. If you want installations to happen on Sundays between 6 p.m. and 8.30 p.m. (duration of 2.5 hours) you would specify

```
SUN-1800-0230
```

Technically this is achieved by using the standard Microsoft Windows mechanism to create a „Scheduled Task". The task is scheduled to restart PKGShell when the next maintenance window is reached. When PKGShell is restarted by the task, all packages that were queued with Schedule=1 are executed.

If you would like to use a MaintenanceWindow value that is stored in a different location in the registry, you can specify that path in the above registry key. If your MaintenanceWindow string is stored under HKEY_LOCAL_MACHINE\-YourKeyName\YourValueName, you would set Parameters\MaintenanceWindow to

```
REG:YourKeyName\YourValueName
```

5 Authoring Packages

5.1 Package definition file basics

The package author controls the behavior of PKGShell by editing an INI file called `PKGShell.SMS` or `PKGSHell.INI` in each package's source directory. This file is used

1. by PKGShell to store detailed information about how the package should be executed
2. by SMS to import a PKGShell integrated SMS Package into the SMS database

The syntax used by PKGShell is a superset of Microsoft SMS's Package Definition File Format (extension 'SMS', previously 'PDF'). PKGShell adds certain parameters that are not used by SMS and are only processed by PKGShell.

One package definition file makes up a **Package**. A **Package** can contain one or several **Programs**, which in turn can contain one or several **Commands**. This terminology is equivalent to Microsoft's SMS.

A minimal package definition file usable by PKGShell might look as follows:

```
[Package Definition]
Build = 1
Name = MSOrca
Programs = Install, Uninstall

[Install]
Command1 = msiexec /I %sourcepath%\orca.msi
Command2 = %comspec% /c dir c :\

[Uninstall]
Command1 = msiexec /x %sourcepath%\orca.msi
```

The PKGShell.exe accepts the name of the Program section to execute as a command line parameter, but defaults to 'Install'. Running 'PKGShell.exe Install' or simply 'PKGShell.exe' with the above PKGShell.SMS file would install the MSOrca package.

☞ Please see section 8.1 on page 65 for a reference of available package parameters.

5.2 Splitting your package into commands

As you may have noticed above, each program section can contain several commands. This is comparable to a batch file because the commands are processed one after another according to the command index that is attached to the parameter name: First 'Command1' then 'Command2', 'Command3'… until no more command are found.

Splitting your installation into several commands is generally a good idea because you get the benefit of **milestone reporting**. Whenever you have one step that constitutes an isolated part of your installation you should define it as a separate command because in case of a failure, PKGShell will report the command the error occurred in.

PKGShell terminates the execution and skips further commands in the package when it encounters an error in a command. If Command1 yields an error, Command2 and Command3 will never be executed.

5.3 Testing for conditions

It is very common to query for certain environmental conditions before or after running a package. While you could use a batch file or VBS scripting to do the queries and return the result be running a command – PKGShell provides a much simpler way to use standard tests without the need for an external programming language.

This is good because extra scripts are a source of complexity and errors. With its simple Text/Ini based syntax, PKGShell tests are easy to read and thus enable easy maintenance of packages.

You can specify a test in place of a command by using the syntax

```
Command1 = TEST:mytest
```

You will then need to write a corresponding section to define your test:

```
[Test:mytest]
Small is beautiful = not DiskFreeMB( 600 )
B = FileExist("%windir%\my.exe") or ServiceExist("MySvc")
```

A test is made up of one or several lines in the package definition file. Each line consists of a name and an expression. The name can be any arbitrary string. Each expression is one line in the package definition file. If one expression evaluates to FALSE, the test is considered failed.

☞ See section 6.2 on page 47 for a complete reference of available keywords in test expressions.

If a test fails, it will include the detailed reason inside the status detail. So the above package might give you the status

```
FAILED:TEST:mytest Small is beautiful (disk has 800/600 MB)
```

To validate the syntax of your expressions while writing a test expression, it's best to let PKGShell start your package with parameter „-P" which will not execute the package. However, a package containing a syntax error will not be queued or run. Instead PKGShell will return the section and key name of the expression that could not be parsed.

Although you could define any number of tests and define each as a separate command, there are three test section names that are predefined and have a special meaning. If you define them in a package definition file they are automatically evaluated at certain stages:

TEST:Success	After the execution of the entire package. If it fails, the final package status will be FAILED.
TEST:PreRun	Just before the package is executed. Status will be CANCELED on failure.
TEST:PreQueue	Is run before the package is accepted into the queue. If it fails, the package is not run and the status will be CANCELED.
TEST:Required	Is run before the package is accepted into the queue and again before it is executed. If it fails, the package is silently skipped. This will ***not*** be reported as an error!

The predefined sections will be evaluated regardless of which 'Program' section is being executed, be it the default 'Install' section or a user-defined one. If you don't want to use a predefined test for a particular program-section, you can override the predefined test-section by appending your program-section name to the test-section name.

This is particular helpful for deinstallations, where you typically do not want to check for the same conditions as with an installation:

```
[Install]
Command1 = msiexec /I my.msi

[Test:PreRun]
Free_Space = DiskFreeMB( 500 )

[Uninstall]
Command1 = msiexec /x my.msi
```

```
[Test:Reqired:Uninstall]
My_File = FileExist("%ProgramFiles%\super.exe" )
```

A test section may also be empty and contain no expressions at all. An empty test always evaluates to TRUE.

If you need to specify a test that explicitly decides whether a certain command was successful or not, it is best to use the parameter property `.Success`:

```
CommandN.Success = Test-Expression
```

E.g. to check whether Command1 really deleted a file you would say

```
Command1         = %comspec% /c del c:\myfile
Command1.Success = Not FileExist("c:\myfile" )
```

If the .Success expression fails, further execution of the package is aborted.

5.4 How commands talk back to PKGShell

PKGShell calls all commands sequentially when it runs a package program and reports a final status for the package after the execution has completed. Each single command that is executed can be a complex program with built-in intelligence to react on special conditions. How can PKGShell leverage this intelligence to enhance its own status reporting? The commands need to talk back and tell PKGShell of any problems they encountered. This is possible through the use of three different techniques:

1. Return Codes

The most basic approach is for the command to return an error code upon termination. This is sometimes referred to as an ‘error level’ because batch files access this code through the environment variable %ERRORLEVEL%. Return codes are widely used by console mode utilities. The error number commonly is a well-known NT error code that was returned by a failed operating system call.

A return code of 0 (zero) means success; any other number is by default regarded as a failure.

If Command1 terminates with an error code of ‘5’ then the reported status of the package will be ‘FAILED’, the status detail will be ‘1:RETURN_ERROR#5’.

If a command uses some different (non-zero) return codes to indicate success you can define this using the *.SuccessCodes* directive:

```
Command1              = %comspec% /c del c:\myfile
Command1.SuccessCodes = 0 2 3 0x80320017
```

If the *.SuccessCodes* directive is not used, the following codes are by default used to indicate success:

```
0                   NO_ERROR
17025               Patch already installed
17026               Patch already admin installed
17028               Patch did not find product
17031               Patch invalid baseline
0x00240006          WSUS_ALREADY_INSTALLED
0x00240007          WSUS_ALREADY_UNINSTALLED
0x80240017          WSUS_E_NOT_APPLICABLE
0x8007f07a          STATUS_SUCCESS_NOREBOOTNEC
0x8007f07b          STATUS_UNINST_NOREBOOTNEC
```

The return code of the last run command is always saved in the variable %LASTERRORLEVEL%. It can be accessed by tests or in subsequent commands – until it is overwritten by the next external command.

2. %ERRORFILE%

In case you are the author of the executable or batch file that is being called as a command, you can give a more verbose feedback for an error like this:

PKGShell defines an environment variable %ERRORFILE%, which holds the name of a non-existent text file. If you encounter an error, you can create that text file – PKGShell will then notice the file was created and will treat that as an error indication. It will also use the content of the file as a description of the error.

PKGShell will parse and process the first 512 characters of the file. The file can be in Unicode or ANSI text format; control characters below ASCII 32 are ignored.

If Command1 executes something like ‘echo bad luck>%ERRORFILE%’ then the reported status of the package will be ‘FAILED:1:bad luck’.

3. Status MIF

A lot of modern setup programs report their status in a so-called ‘Installation Status MIF’ file. This is a specially formatted text file that reports the status (success or failure) along with an error description. Setup engines like the Windows Installer make an effort to give an intelligent explanation of what went wrong, so this description is usually very efficient for troubleshooting.

PKGShell will by default instruct the Windows Installer to generate a MIF file called %MIF% in the %TEMP% directory by using the MSIEXEC parameter „/M". You could of course choose to generate a MIF file with the same name using any mechanism different from MSIEXEC. If however the MIF file is found present after a command completes, PKGShell will parse its content to get the status and the description of an error contained therein.

If the first command (Command1) generates an error MIF file with an error description of 'Bad Luck', then the reported status of the package will again be 'FAILED', the status detail will be '1:Bad Luck'.

If you are using commands that have no built-in intelligence and arbitrarily return meaningless error codes, you can set the PKGShell parameter property „.IgnoreError=1". This will make PKGShell ignore the return codes of a particular command.

```
Command1             = %comspec% /c del c:\myfile
Command1.IgnoreError = 1
```

This can also be helpful if you simply do not care if a specific command in your package fails or not – since it may be irrelevant to the overall success of the package.

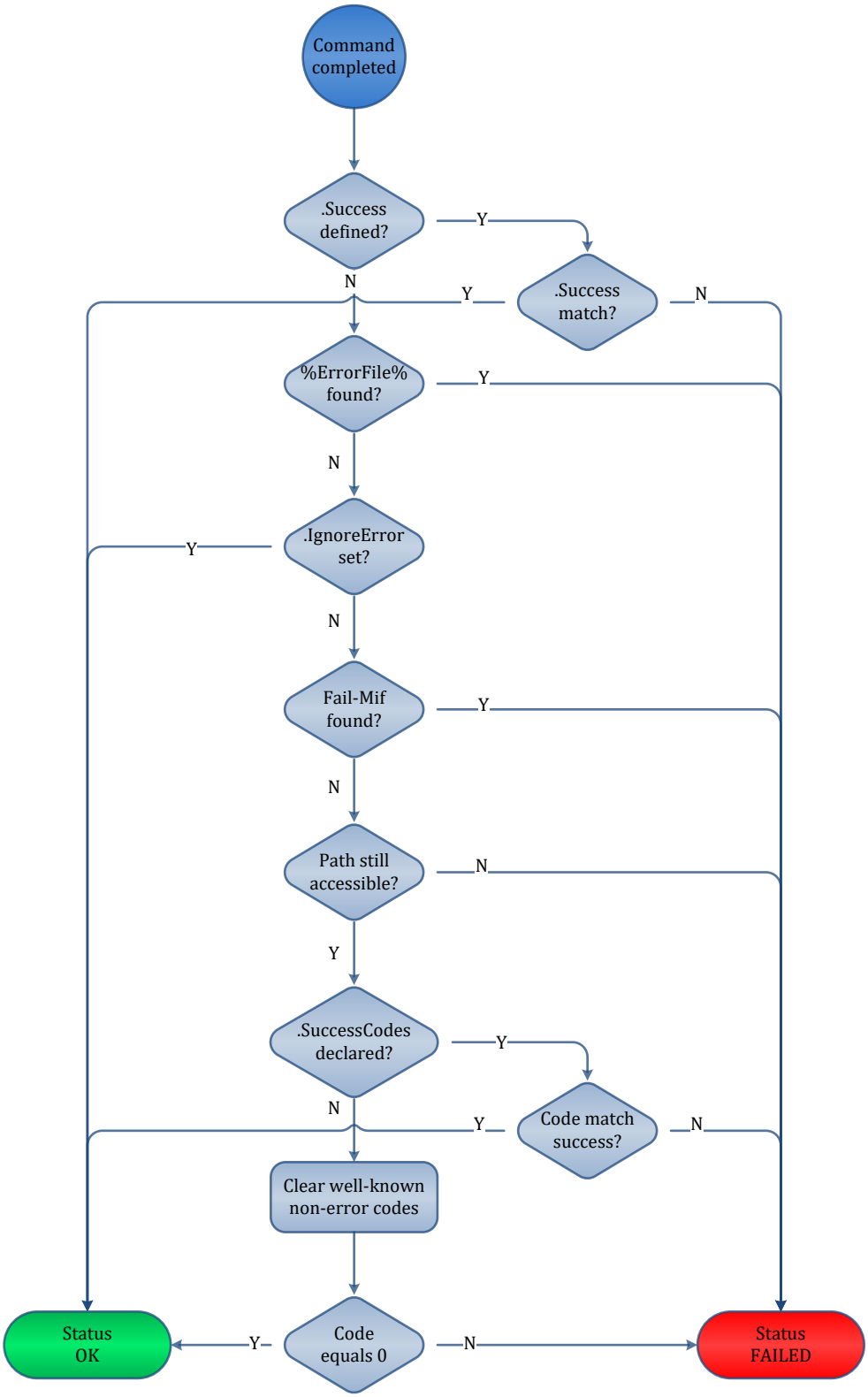
If you use the above parameter because your external programs are incapable of reporting success or failure, you should use the PKGShell Command.Success directive or a TEST:Success section to define some basic criteria of success for your package.

Each of the above techniques can be used independently or in combination. Beware that PKGShell has preferences over which technique will override the other when deciding about a command's status:

1. Perform test as defined by the .Success Parameter.
2. Check for a MIF file and parse it. If it's a Fail-MIF, report the description as status 'FAILED' with detail 'X:DESCRIPTION'
3. Check for the existence of %ERRORFILE%. If it is found, report the content as status 'FAILED' with detail 'X:ERRORFILE-CONTENT'
4. Make sure the package source path is still accessible after execution. If it's inaccessible, report status 'ABORTED' with detail 'X:PATH_ERROR#NUMBER'
5. Test for a non-zero return code. If it's non-zero, report status 'FAILED' with detail 'X:RETURN_ERROR#NUMBER'
6. Otherwise, return 'OK'

The idea is to give a higher priority to the supposedly more intelligent error explanations. E.g. if a MIF file is written, the non-zero return code will not be reported, because a MIF file is more *meaningful*. If an error file is created, no network failure will be reported, because error files contents are usually more meaningful.

An overview of this decision process is explained by the flow chart on the following page:



5.5 PKGShell environment variables

A number of environment variables are defined by PKGShell at runtime. They are available to be accessed by the package commands:

```
%BUILD%           the package build as defined
%ERRORFILE%       full path to a file used for error control
%LASTERRORLEVEL% return code of last extermal command
%LOGDIR%          full path to the log file directory
%LOGFILE%         full path to the package's log file
%MIF%             an 8 char file name unique to the process
%NAME%            the package name as defined
%OSARCH%          x86 or x64
%OSBITS%          32 or 64
%OSLANG%          i.e. ENU for english
%OSLANGCODE%      i.e. 1033 for english
%OSBUILD%         i.e. 17763 for Windows 10 1809
%OSVER%           i.e. 0602 for Windows 8
%SOURCEPATH%      full (UNC) path to the package directory
```

Note: when installing MSI packages, you should use the variable %SourcePath% to specify the path to the MSI, because this path will be remembered for later access by the Windows Installer. If you used a relative path, it may contain a temporarily redirected drive letter that is not available after execution. %SourcePath% is always in UNC format, unless the source is local.

The current directory when executing a command is always set to the package root alias %SourcePath%. You can override this behavior by using the command parameter CD:

```
Command1.Desc     = Show contents of temp dir
Command1          = %comspec% /c dir
Command1.CD       = %temp%
```

For running 32bit software on 64bit systems there are a number of variables that can save complexity when building portable packages. These variables consistently point to the 32bit locations of system areas:

```
%32bit_ProgramFiles%    32bit:      %ProgramFiles%
%32bit_ProgramFiles%    64bit:      %ProgramFiles(x86)%
%32bit_System32%        32bit:      System32
%32bit_System32%        64bit:      SysWow64
%32bit_Software%        32bit:      Software
%32bit_Software%        64bit:      Software\Wow6432Node
```

5.6 Understanding reboots

A reboot that is initiated by PKGShell is called an internal reboot. A reboot done by the command is called an external reboot. PKGShell knows 4 ways to deal with reboots of package commands:

1. **Dynamic Internal** (Default): Commands request a reboot by returning a special return code to PKGShell whenever they need a reboot.
2. **Suppressed**: PKGShell will not honor dynamic reboot requests.
3. **Hardwired, Internal**: PKGShell will always reboot after a certain command.
4. **External**: PKGShell will tolerate that the command may perform a reboot by itself.

> ☞ As a guideline, no command should directly reboot the computer by itself. If a command requires a reboot, it should instruct PKGShell to do so. By default, PKGShell will treat an external reboot attempt from a command as an error condition and will report a status of 'ABORTED:SHUTDOWN'.

5.6.1 Dynamic (Default)

By default, PKGShell will not reboot after executing a command. However, the command can optionally request a reboot by returning a special return code.

Using a return code is the preferred method for requesting a reboot because it is dynamic – the command is able to request a reboot on an as-needed basis.

The error codes that can be returned by a command and which will cause PKGShell to perform a reboot are:

```
1641        ERROR_SUCCESS_REBOOT_INITIATED
3010        ERROR_SUCCESS_REBOOT_REQUIRED
3011        ERROR_SUCCESS_RESTART_REQUIRED
17022       Office patch reboot required
0x00240005  WSUS_REBOOT_REQUIRED
0x80070BC2  STATUS_REBOOT_REQUIRED
0x8007F008  STATUS_UPDATE_SUCCESSFUL
0x8007f075  STATUS_SUCCESS_NOREBOOT
```

These error codes are commonly used by the windows installer and are automatically returned by MSIEXEC.EXE when it performs an installation and wishes to reboot.

5.6.2 Suppressed

If you do not want PKGShell to perform a reboot although the command has requested so (by returning one of the above return codes) you can use the parameter `SuppressReboot`.

```
Command3.SuppressReboot = 1
```

The execution will continue with the next command even if a reboot was requested. Beware that the subsequent commands may not behave as expected while a reboot request is pending.

This method can give a significant speed advantage if several commands request a reboot.

A reboot request that was suppressed by one command in the package is by default performed after all commands in the package have completed – so the reboot is really only postponed. To completely prevent rebooting, see "final" reboots (section 5.6.5).

5.6.3 Hard-wired

If you prefer a static solution that will make PKGShell always perform a reboot after a certain command, you can use the parameter `DoReboot`.

```
Command3.DoReboot = 1
```

Specifying the above parameter in the Program section of a package definition file will cause an unconditional reboot to be performed after Command3.

5.6.4 External

If you have a „misbehaved" command that insists on performing a reboot itself, you need to use the `TolerateReboot` parameter.

```
Command3                = setup.exe /reboot
Command3.TolerateReboot = 1
```

If e.g. a setup program reboots the computer and you have **not** specified `TolerateReboot`, PKGShell will report an error status of `ABORTED:SHUTDOWN`, because the external reboot was unexpected.

5.6.5 Final

If any reboots requested by commands in your package have been suppressed using `SuppressReboot=1`, then PKGShell will perform a dynamic reboot after the last package command has been processed.

To also suppress this postponed reboot, you can use the package parameter

```
SuppressFinalReboot = 1
```

5.6.6 Retry

There is another case that needs special attention – Windows uses certain error codes to indicate that the installer cannot continue without first performing a reboot.

```
1604         ERROR_INSTALL_SUSPEND
0X8007F02A   STATUS_MUST_RESTART_FIRST
```

PKGShell will detect this request and dynamically perform a reboot. After the reboot, the same command is executed again (retried).

If the command returns the same code on the second run, this will be reported as an error. A reboot-retry is not performed more than once to prevent endless loops.

5.7 Grouping commands with sub-routines

If you have a number of commands that you have to repeat in more than one section inside the same package file you may group these commands inside a subroutine call:

```
[Install]
Command1 = %comspec% /c echo Before sub call
Command2 = SUB:myname
Command3 = %comspec% /c echo Back from sub call

[Install_Otherwise]
Command1 = SUB:myname

[SUB:myname]
Command1 = %comspec% /c echo In sub call
Command2 = %comspec% /c cd
```

After the sub is processed, the execution continues with the next command after the call. A sub call can include other sub calls. As usual, be careful not to create endless loops.

> ☞ Note: A subroutine section contains only commands and command properties. Unlike a regular program section, it cannot contain package directives like Logoff=1 etc.

There is one special sub-section called ‘Finally’ that, if defined, is always executed after the main section has finished. This section is particularly helpful for doing cleanup actions that must always be executed – because it is always executed, regardless of whether the whole package completed or whether it was interrupted by an error:

```
[Install]
Command1.Desc = Disable Bitlocker protection
Command1 = cscript manage-bde.wsf –protectors –disable C:
Command2 = SUB:BIOS-Update
```

```
Command3 = apply_bios_settings.exe

[Install:Finally]
Command1.Desc = Enable Bitlocker protection
Command1 = cscript manage-bde.wsf –protectors –enable C:

[SUB:BIOS-Update]
Command1 = flash_bios.exe
Command1.DoReboot = 1
```

In the above example, if the re-enabling of the protection had not been put in a 'Finally'-section, the computer would be left unprotected if the BIOS upgrade failed.

> ☞ Note that the 'Finally' section behaves like a sub, although its name does not start with the prefix 'SUB:'. Instead, it is prefixed with the name of the main section that it belongs to.

5.8 Controlling the flow of execution

The commands of a package/program are executed sequentially, one after the other. If one command reports an error, all further commands (except those in the 'Finally' section) are skipped. This is a simple approach that implies all commands must be executed successfully as part of the installation.

5.8.1 Required

If you want to execute a command only when a certain condition is met, PKGShell supports a parameter `Required` that does just that:

```
CommandN.Required = Test-Expression
```

So e.g. if Command1 needs to be executed only when a file is present you would say

```
Command1          = %comspec% /c del c:\myfile
Command1.Required = FileExist("c:\myfile" )
```

5.8.2 Until

To repeat a command **until** a condition is met, use the `CommandX.Until` directive:

```
Command1       = %comspec% /c del c:\myfile
Command1.Until = not FileExist("c:\myfile" )
```

5.8.3 While

Similarly, to repeat **while** a condition is true, use the `CommandX.While` directive:

```
Command1       = %comspec% /c del c:\myfile
Command1.While = FileExist("c:\myfile" )
```

Of course, the above examples are somewhat naïve and should not be used in a production environment, since they may result in an endless loop.

If several commands should be executed as a loop you can control the execution of a sub:

```
Command1       = SUB:KillFile
Command1.While = FileExist("c:\myfile" )
```

5.8.4 Foreach

To let a command loop through the values of an array you can use the Foreach property:

```
Command1         = reg.exe add HKLM\Software\%_% /f
Command1.Foreach = test, "my other company"
```

This will actually execute Command1 two times and create both registry keys “HKLM\Software\test” and “HKLM\Software\my other company”.

If you prefer to use a named variable instead of the implicit %_% the same example would read:

```
Command1           = reg.exe add HKLM\Software\%i% /f
Command1.Foreach:i = test, "my other company"
```

Foreach arrays are automatically expanded for certain wildcards. See the following examples:

```
Command1.Foreach = %temp%\*.log                  (files)
Command1.Foreach = HKLM\Software\*Company*        (registry)
Command1.Foreach = PKG:*Flash*                   (packages)
Command1.Foreach = MSI:*Flash*                   (products)
Command1.Foreach = %{ WmiValue( "Win32_Volume.Name" ) }%
```

Entries in the array can be separated by <space> <comma> or <tab> characters. If you use <tab>, this cannot be mixed with <space> or <comma>.

Automatically expanded arrays are internally separated by <tab>.

5.8.5 Skipnext

If a command needs to decide at runtime whether the *next* single command should be skipped, the command can either

1. write the string ‘SKIPNEXT’ into the %ERRORFILE% or

2. return the error code 1246 (ERROR_CONTINUE)

PKGShell will then skip the execution of the immediately following command.

5.8.6 Exit

The `EXIT` command terminates execution immediately, skipping all further commands. If used inside of a SUB-Block, the block is terminated but execution is continued with the next command after the Sub-Call – so it's actually a „return".

```
CommandN = EXIT
```

5.8.7 Goto

The `GOTO` command does an unconditional jump to a specified command and can use either a command number as a target like:

```
CommandN = GOTO Command#
```

The `GOTO` command can also specify a label of another command as a target:

```
Command3          = GOTO Mylabel
Command4          = skipped.exe
Command5.Label    = Mylabel
Command5          = target.exe
```

5.8.8 If … Goto

More flexibility can be obtained by using the conditional jump instruction:

```
CommandN = IF : Test_Expression : Command#
```

So, e.g. to jump to the 5th command if a file does not exist, you say

```
Command1 = IF : Not FileExist("c:\boot.ini" ) : 5
…
Command5 = …
```

Or, using a label:

```
Command1 = IF : Not FileExist("c:\boot.ini" ) : no_ini
…
Command5.Label = no_ini
Command5 = …
```

The test expression use the same syntax as mentioned earlier for the test sections. See section 6.2 on page 47 for a complete reference.

If you need more complex control you may prefer to put this intelligence into a separate script and call that script as a command. PKGShell is not designed as a substitute for a full programming language – it just minimizes the need for external scripting in standard scenarios.

5.9 Grouping packages with sub-packages

You may create a package that contains one or several other packages that need to be executed together as one. This has the advantage that the components might also be executed independently. The installation status of each component will be reported as an individual package.

The PKG: keyword on a command line of the parent package invokes a sub-package:

```
Command1 = PKG:Package2
```

How can PKGShell locate the path to other packages? Particularly in an SMS environment, the directory names of each package are not known and may vary from server to server. That's why the name 'Package2' does not reference a directory name, but rather the package name as defined in the package definition file.

The previous example will scan all directories on the same level as the current package's directory for a „sister" directory that contains a package definition file that defines a package name of 'Package2'. The above example is actually equivalent to

```
Command1 = PKG:..\Package2
```

If you want to specify a different path to search, you would say:

```
Command1 = PKG:%temp%\Package2
```

If you want to invoke a specific program/section of the sub-package instead of the default 'Install' section, you have to specify it by appending the section name to the package name like so:

```
Command1 = PKG:PackageName:SectionName
```

The execution of a sub-package takes place immediately. This is a similar to a function call – execution will continue with the next command only when the sub-package call is completed and control returns to the current package. Failure to execute the sub-package is equivalent to a command failure.

If you want to ignore the resulting status of a sub-package, you can use the parameter „IgnoreError=" just as you would use it to ignore the result of a single regular command.

☞ Note: The calling of sub-packages with 'PKG:' does not work with the parameter „Direct=1". Attempting to call a sub package when either the parent or the child package uses this parameter leads to an error and the execution is aborted.

5.10 Defining environment variables

To enhance the readability of commands, you can declare commonly used strings as environment variables. This is done by defining a `Strings` section in the package definition file:

```
[Strings]
MyPath = %ProgramFiles%\MyApp
MyParams = /c /s

[Install]
Command1 = %comspec% /c copy *.exe "%MyPath%"
Command2 = '"%MyPath%\my.exe" %MyParams%'
```

The defined strings are expanded into regular environment variables at runtime and can further be used internally by the called commands.

You can also define and modify variables at run time by using the internal command SET:

```
Command1 = SET MyPath=%PATH%;%ProgramFiles%\MyApp
```

PKGShell variables are accessible as environment variables during package execution, but they do not 'exist' after the package is finished. Setting or changing variables in packages does not modify the computer's system environment.

☞ Note: The parameter `NoExpand=1` can be used to prevent the expansion of variables when assigning.

5.11 Using PKGShell expressions in commands

You can use PKGShell expressions as part of your commands: the special string `%{`*`expression`*`}%` will be evaluated by PKGShell before your command is run.

The expressions that you can use are a subset from the expressions that can be used for testing conditions. See section 6.1 on page 45 for all string functions available.

To print the Windows version number to your log file, you could write:

```
Command1 = echo %{
     regvalue("HKLM\Software\Microsoft\Windows
     NT\CurrentVersion\CurrentVersion" ) }%
```

5.12 Using the CLEAN command to delete old objects

If you find yourself writing a lot of code to cleanup leftovers from older packages, you might consider using the internal CLEAN command.

```
Command1 = CLEAN pkg:name* msi:*name* hklm\name*
     %temp%\*.tmp
```

The CLEAN command uses the Foreach command property to loop through wildcards. So the above is technically equivalent to saying

```
Command1 = CLEAN %_%
Command1.Foreach = pkg:name* msi:*name* hklm\name*
     %temp%\*.tmp
```

This spares you from checking for existence each time before trying to delete an object. It supports wildcards and the following object types:

5.12.1 Packages

Identified by the prefix “PKG:” any local packages that are found installed are enumerated. If a package name matches the given wildcard, the uninstall (or deinstall) section is called as a sub PKG call.

```
Command1 = CLEAN pkg:*flash*
```

Will call the uninstall section of all packages with “flash” in their names.

```
Command1 = CLEAN pkg:*flash*:kill
```

Will call the section named “kill” for the same packages. This is equivalent of saying

```
Command1 = pkg:%_%:kill
Command1.Foreach = pkg:*flash*
```

5.12.2 Windows Installer products

Identified by the prefix “MSI:” any msi products that are found installed are enumerated. If a product name matches the given wildcard, the product code is uninstalled.

```
Command1 = CLEAN msi:*flash*
```

Will call the uninstall section of all packages with “flash” in their names. This is equivalent of saying:

```
Command1 = msiexec.exe /x %_%
Command1.Foreach = msi:*flash*
```

5.12.3 Registry paths

Registry paths are identified by one of the following common prefixes:

```
HKLM or HKEY_LOCAL_MACHINE
HKU or HKEY_USERS
HKCU or HKEY_CURRENT_USER
HKCR or HKEY_CLASSES_ROOT
HKCC or HKEY_CURRENT_CONFIG
```

For example,

```
 Command1 = CLEAN hklm\software\*name*
```

Will delete all subkeys with “name” in their names. This is equivalent of saying:

```
Command1 = reg.exe delete %_% /force
Command1.Foreach = hklm\software\*name*
```

5.12.4 File paths

File paths given can be either files or directories that both will be deleted if they exist. For example:

```
 Command1 = CLEAN %ProgramFiles%\*name*
```

5.13 Customizing the package dialog

Each package that has „Logoff=1” should provide an informational HTML page that describes the package to the user. When PKGShell prompts the user to logoff, the user can display and browse through each pending package’s info page.

To create an info page for a package, you need to author an HTML file that can contain any valid HTML with links, images and the like. After the page is ready, you must convert it to a MIME-encoded HTML Document (MHTML) with the extension .MHT.

This single file will contain formatting, background and embedded pictures. Microsoft Internet Explorer can easily create such a file if you open your page and then choose „File" / „Save as…" and select a type of „Web Archive, single file".

The name of the file must be

```
info_<program>_<language>.mht
```

The language name used is the 3-letter ISO standard abbreviation. So for example

```
info_install_enu.mht
```

will be displayed if the „Install" section has been queued and the logged on user's language is US English.

All info .MHT files you create for a package must be placed in the same directory as the package definition file.

5.14 Preparing packages for use with SMS

The above PKGShell.SMS examples are fully functional for running PKGShell without SMS. Actually it is sufficient to use them with SMS as described above if you manually create an SMS package through the SMS console and configure the correct parameters for the package program to execute PKGShell. But because this manual process is error prone and the SMS operator may not have enough knowledge about the correct parameters for calling PKGShell.exe, the package definition file can be used to import and create the package in SMS.

In order to make a package definition file viable for importing into SMS, certain standard parameters (marked yellow below) must be included:

```
[Package Definition]
Build = 1
Name = MSOrca
Programs = Install, Uninstall
Publisher = Microsoft
Language = English

[Install]
Command1 = msiexec /I %sourcepath%\orca.msi
Command2 = %comspec% /c dir c:\
Name = Install
CommandLine = pkgshell.exe install
AdminRightsRequired = True
CanRunWhen = AnyUserStatus
Run = Hidden
SupportedClients = Win NT(i386)
```

```
[Uninstall]
Command1 = msiexec /x %sourcepath%\orca.msi
Name = Uninstall
CommandLine = pkgshell.exe uninstall
AdminRightsRequired = True
CanRunWhen = AnyUserStatus
Run = Hidden
SupportedClients = Win NT(i386)
```

5.15 Debugging packages

To validate a package definition for correct syntax, you can either try to execute it or you can use the command line parameter „-p" (Pre-Run-Check).

After you have started the execution the main source of information is the package log file that by default is under `%WINDIR%\Logs\<package>.log`. It is advisable to open this log with a tool like SMS Trace32.exe to monitor the current activity.

During the execution of a package with Logoff=1 there are special keys you can use to get more control than a normal user:

5.15.1 Temporary Re-Enabling the Logon Dialog

You can stop PKGShell from hiding the logon dialog by simultaneously holding down both the left and the right shift keys. The logon dialog (CTRL-ALT-DELETE) will be shown again. If you press <LEFT SHIFT> + <RIGHT SHIFT> again, the logon dialog will be hidden again.

5.15.2 Opening a debug console

You can open a command prompt in the same context as the executing package to interactively test commands. If you simultaneously press <LEFT CTRL> + <RIGHT CTRL>, you will be prompted for a user name and password of an account that has to be member of the local administrator group. If the credentials are correct, you get a console window running as SYSTEM.

5.15.3 Cancelling further execution

If you press <CTRL> + C or <CTRL> + <BREAK> on the progress dialog, execution of the queued packages will stop after the current package has finished running.

This can be also triggered from the command line with the parameter "-break".

If the current package fails and a retry would be performed, the retry is **not** performed if you issued a break.

The other packages currently in the queue stay in the queue. But because the consent is taken away, the package dialog will re-appear to confirm further execution.

6 Language

This section serves as a reference to the language used by PKGShell tests.

> ☞ You can verify correct syntax by running a pre-check (-p) on the package

6.1 Syntax specification

bexpression	:=	bterm [**OR** bterm]*
bterm	:=	bfactor [**AND** bfactor]*
bfactor	:=	**(** bexpression **)** **NOT** bfactor bfunction equation
bfunction	:=	**DiskFreeMB(** string **)** **FileExist(** string **)** **OsAtLeast(** string **)** **RegExist(** string **)** **ServiceExist(** string **)** **SubNet(** string **)** **WmiExist(** string **)** **WmiExist(** string**,** string**)**
equation	:=	string **=** string string **>** string string **>=** string string **<** string string **<=** string string **<>** string
string	:=	sfunction literal
sfunction	:=	string **+** string **Concat(** string, string **)** **DateAdd(** string, int, string **)** **DateDiff(** string, string, string **)** **FileContent(** string [, int] **)** **FileDate(** string **)** **FileVersion(** string **)** **FileVersionNumber(** string **)** **Find(** string**,** string **)** **IniValue(** string**,** string**,** string **)** **Left(** string, count **)** **Len(** string **)** **MsiProductStatus(** string **)** **Now** **RegValue(** string **)** **Right(** string, count **)** **ServiceStatus(** string **)** **Switch** string**:** [string**:**string]* **else:**string **WmiValue(** string **)**
literal	:=	**"..[""].."** **'..['']..'** [**-0123456789**] **0x**[**0123456789ABCDEF**]

The expressions are case **in**sensitive, all keywords and parameters are internally converted to lowercase before further processing.

Literal text should be enclosed in either single (') or double (") quotation marks. If you want the string to include the same quotation marks that it is enclosed by, you must 'escape' them by specifying two quotation marks for every single appearance. So <Say 'Hello'> should be quoted as <'Say "Hello"'>.

Beware that there is a difference in using **single** or **double** quotation marks – this is equivalent to string literals in the Perl scripting language: When using double quotation marks, environment variables within the strings are expanded.

That means that `FileExist("%windir%\test")` will expand `%windir%` in the string (i.e. `to "c:\winnt\test")`, but `FileExist('%windir%\test')` will probably not do what you want. Generally, you should stick with double quotation marks; because they let strings work the way everybody expects them to. Only when you run into a case where you want to literally use a variable name, you can fall back to single quotation marks.

When comparing values with operators (=, < or >) special care is taken to consider version numbers:

```
"9.1" < "10.0"
"1.0.13" > "1.0.3"
```

6.2 Keyword reference

Following is an alphabetical list of all valid function keywords:

6.2.1 Concat(string1, string2)

Append string2 to string1 and return the result.

Example:

```
Concat("AB", "CDE" ) = "ABCDE"
"AB" + "CDE" = "ABCDE"
```

6.2.2 DateAdd(interval, number, date)

Returns a WEBM date string with the specified number of intervals added:

```
          ->       set date = "20150101000000.000000+000"
DateAdd( 's', 1, "%date%" ) = "20150101000001.000000+000"
DateAdd( 'n', 1, "%date%" ) = "20150101000100.000000+000"
DateAdd( 'h', 1, "%date%" ) = "20150101010000.000000+000"
DateAdd( 'd', 1, "%date%" ) = "20150102000000.000000+000"
DateAdd( 'm', 1, "%date%" ) = "20150131102902.397120+000"
DateAdd( 'y', 1, "%date%" ) = "20160101054828.791360+000"
DateAdd( 'y',-1, "%date%" ) = "20131231181131.208640+000"
```

6.2.3 DateDiff(interval, date1, date2)

Returns an integer as the number of intervals between date1 and date2. The dates have to be in WBEM format:

```
DateDiff( 'd', "20150101000000.000000+000",
      "20150102000000.000000+000" ) = 1
```

Calculate the number of minutes since the computer was started:

```
DateDiff( "n", WMiValue(
      Win32_OperatingSystem.LastBootUpTime ), now )
```

6.2.4 DiskFreeMB(count)

Will verify that the system partition –the one which holds %SystemRoot%– has the specified amount of available free space.

Example – Check if the disk has at least 250 MB free:

```
DiskFreeMB( 250 )
```

Will return TRUE on success.

6.2.5 FileContent(path [, lines])

Retrieve a specified file's content as a tab-separated string array. The return value can be used in a ForEach loop to iterate over a command.

Example 1 – Delete all files listed in a text file:

```
Command1 = %comspec% /c del %_%
Command1.Foreach = FileContent( "%temp%\list.txt" )
```

Example 2 – Check if the last two lines of a file contain the word 'Hello':

```
Find( FileContent( "%temp%\list.txt", -2 ), "Hello" ) > 0
```

6.2.6 FileDate(path)

Retrieve a specified file's modification time (a.k.a. last write) as a DMTF date string.

Example – Verify version of cmd.exe:

```
DateDiff('h', FileDate("%windir%\windowsupdate.log"), now)
      < 24
```

6.2.7 FileExist(path)

Test if a file or a directory exists. Wildcards are supported.

Example – Verify the existence of a program's .exe:

```
FileExist("%ProgramFiles%\application\main.exe" )
```

Example – Verify the existence of a drive:

```
FileExist("e:\" )
```

6.2.8 FileVersion(path)

Retrieve a specified file's version as a string.

Example – Verify version of cmd.exe:

```
FileVersion("%windir%\system32\cmd.exe" ) >= "5"
```

6.2.9 FileVersionNumber(path)

Retrieve a specified file's **numeric** version number as a string. The numeric version number always consists of 4 words separated by a dot. This is often more consistent than the string returned by `FileVersion()`, because the format of the contained text is predefined. Some executables report string versions as '*3.0a Servicepack 2*', which is not as easy to compare as '*3.0.1.312*'.

Example – Verify that the cmd.exe on your system is at least of XP RTM:

```
FileVersionNumber("%windir%\system32\cmd.exe" ) >=
      "5.1.2600.0"
```

6.2.10 Find(string, substring)

Try to find the first occurrence of a substring inside of a string. If the substring is found, the function returns the index of the character on which the substring starts. If the substring is not found, it returns 0 (zero).

Example:

```
Find("Abcde", "B" ) = 2
Find("ABCDE", "Z" ) = 0
```

6.2.11 IniValue(path, section, key)

Retrieve a value from an .INI file. The function takes 3 parameters: The path name to the ini file, the section name and the key to retrieve.

Example – Compare the wallpaper:

```
IniValue("%windir%\win.ini", "desktop", "wallpaper" ) <>
      "pamela.bmp"
```

6.2.12 Left(string, count)

Returns a specified number of characters from the left side of a string. Example:

```
Left("ABCDE", 2 ) = "AB"
```

If count is negative, then cut off the specified number of characters from the end:

```
Left("ABCDE", -2 ) = "ABC"
```

6.2.13 Len(string)

Returns the number of characters in a string.

Example:

```
Len("ABCDE" ) = 5
```

6.2.14 MsiProductStatus(code)

Returns the installation state of a product installed by the Windows Installer. The state is returned as a string literal that can be compared to a specified value.

Example – Check if the Windows XP Support Tools are installed:

```
MsiProductStatus("{8398B542-3CC4-44D9-83DF-696CCE70124B}" ) = 5
```

The states that this function can return are documented in the Windows Installer SDK:

Number	Status	Explanation
-7	**NOTUSED**	component disabled
-6	**BADCONFIG**	configuration data corrupt
-5	**INCOMPLETE**	installation suspended or in progress
-4	**SOURCEABSENT**	run from source, source is unavailable
-3	**MOREDATA**	return buffer overflow
-2	**INVALIDARG**	invalid function argument
-1	**UNKNOWN**	unrecognized product or feature
0	**BROKEN**	broken
1	**ADVERTISED**	advertised feature or being removed
2	**ABSENT**	uninstalled or installed for different user
3	**LOCAL**	installed on local drive
4	**SOURCE**	run from source, CD or net
5	**DEFAULT**	installed for the current user, local or source

6.2.15 Now

Returns the current UTC date and time as a WBEM date string:

```
Now = "20150120215301.971674+000"
DateDiff( "n", now, DateAdd( "s", 80220, now ) ) = 1337
```

6.2.16 OsAtLeast(name)

Verify that the Windows operating system build is equal or greater than the given name's build.

The given name is matched against the following list of known Windows build names, first match wins:

```
2019
1809
1803
1709
1703
2016
1607
1511
1507
windows 10
windows 8.1 update 1
windows 8.1
2012r2
windows 8
```

```
2012
windows 7 update 1
2008r2 update 1
windows 7
2008 r2
vista update 2
2008
vista
2003
xp
2000
nt4
```

Example – Check for Vista or greater:

```
OsAtLeast( "Vista" )
```

6.2.17 PackageStatus(name)

Returns the status of a package installed by PKGShell. The name parameter is the name of the package as specified in its definition file.

Example – Verify a previous successful execution:

```
PackageStatus("MSOrca" ) = "OK"
```

6.2.18 RegExist(path)

Test if a registry key exists.

Example:

```
RegExist("HKLM\Software\Microsoft" )
```

Will return TRUE on success.

6.2.19 RegValue(path)

Return the content of a registry value as a string. The name of the value to be read must be concatenated to the path of the key.

The path string must begin with one of the following root entries:

```
HKLM or HKEY_LOCAL_MACHINE
HKU or HKEY_USERS
HKCU or HKEY_CURRENT_USER
HKCR or HKEY_CLASSES_ROOT
HKCC or HKEY_CURRENT_CONFIG
```

Examples:

```
RegValue("HKLM\Software\Company\PKGShell\Status" ) = "OK"
RegValue("HKLM\Software\Company\PKGShell\Build" ) < 0xFF
```

6.2.20 ReturnCode(command)

Process the error level (return code) of an external command.

Example – The following should always be true…

```
ReturnCode("%comspec% /c exit 13" ) = 13
```

Test if a COPY command fails:

```
ReturnCode("%comspec% /c copy e:\test c:\test" ) > 0
```

6.2.21 Right(string)

Returns a specified number of characters from the right side of a string.

Example:

```
Right("ABCDE", 2 ) = "DE"
```

If count is negative, then cut of the specified number of characters from the start:

```
Right("ABCDE", -2 ) = "CDE"
```

6.2.22 ServiceExist(name)

Test if a service as installed.

Example – Check if the PKGShell service exists:

```
ServiceExist("pkgshell" )
```

6.2.23 ServiceStatus(name)

Return the state of an installed service as a string.

Example – Check if the PKGShell service is stopped:

```
ServiceStatus("pkgshell" ) = "stopped"
```

The possible status strings returned are:

Status	*Explanation*
STOPPED	The service is not running
START_PENDING	The service is starting
STOP_PENDING	The service is stopping
RUNNING	The service is running
CONTINUE_PENDING	The service continue is pending
PAUSE_PENDING	The service pause is pending
PAUSED	The service is paused
UNKNOWN	Other

If the service does not exist, an empty string is returned.

6.2.24 SubNet(network)

Check if at least one of the clients's IP addresses lies in a specified IP network. If the client has more than one local IP address, all active addresses are processed and tested.

The network must be specified using the newer ip-slash-mask format. Thus a 16 bit subnet mask is specified as „/16" rather than as „255.255.0.0"

Example – Test for membership in subnet 10.0.0.0:

```
SubNet("10.0.0.0/8" )
```

6.2.25 Substr(string, start, count)

Returns a specified number of characters from a string.

If the start is 0 or 1, the result is equivalent to "**`Left(string,count)`**". If count is larger than the number of remaining characters, the characters up to the end are returned.

Examples:

```
Substr("ABCDE", 1, 2 ) = "AB"
Substr("ABCDE", 0, 2 ) = "AB"
Substr("ABCDE", 2, 2 ) = "BC"
Substr("ABCDE", 3, 6 ) = "CDE"
Substr("ABCDE", 3,-1 ) = "CD"
```

6.2.26 Switch string: [string:string]* else:string

The Switch statement lets you choose one of several strings depending on the values of another string. This is very much like the switch statement in VBScript or Powershell:

```
[Strings]
Always_D = %{ Switch "A": "B":"C" else:"D" }%
DOMENV = %{ Switch WmiValue( "win32_computersystem.domain"
      ):  "MSFT":"PROD" else:"TEST" }%
```

The variable DOMENV above will be set to "PROD" if the system's domain is "MSFT", otherwise it will be "TEST".

WmiExist(class, match)

Returns true if a Wbem class instance is found that matches the given pattern.

The class parameter can optionally include a namespace specification and a property name in the form of „[namespace\]class[.property]". If a namespace is not given, the default of „root\cimv2" is assumed. If the property is omitted, the property „Name" is used by default.

The match value is case insensitive and can include wildcards in the form of „*" (any number of characters) and „?" (a single character).

Test if the computer vendor is „Medion":

```
WmiExist( "Win32_ComputerSystemProduct.Vendor", "MEDION" )
```

Test if the SMS remote tools are installed (using a non-default namespace):

```
WmiExist("root\ccm\CCM_InstalledComponent", "*RemoteTools")
```

Test if the computer contains any ATI graphics card:

```
WmiExist("Win32_PnPEntity.DeviceID", "PCI\VEN_1002&DEV_4*")
```

☞ Note that the performance of the query is faster if you do not use wildcards in the match string. If no wildcards are used, the query is internally translated into a faster WQL query.

If you are familiar with WQL queries you can also use WQL directly by specifying only one parameter to the WmiExist function:

```
WmiExist("select * from Win32_Product where name='X' and
      version='Y'")
```

Or equivalent with namespace:

```
WmiExist("root\cimv2\select * from Win32_Product where
      name='X' and version='Y'" )
```

6.2.27 WmiValue(object)

Returns the content of a Wbem class instance property. This makes most sense when there is only one instance of a class because there will be only one result. If several instances are found, the values are concatenated using a Tab character.

As with WmiExist above, the namespace can be given (defaults to root\cimv2) and the property can be omitted (defaults to “name”).

This example stores the computer vendor in a variable:

```
[Strings]
Vendor = %{WmiValue("Win32_ComputerSystemProduct.Vendor")}%
```

If you know WQL, this example can tell when the computer has been running for over an hour:

```
Command1 = %comspec% /c echo Maybe it's time for a break
Command1.Required = WmiValue("select SystemUpTime from
      Win32_PerfFormattedData_PerfOS_System" ) > 3600
```

Actually, because we’re not storing the value anywhere, the following is equivalent but maybe more efficient:

```
Command1 = %comspec% /c echo Maybe it's time for a break
Command1.Required = WmiExist("select * from
      Win32_PerfFormattedData_PerfOS_System where
      SystemUpTime > 3600" )
```

7 Customization

You can better fit PKGShell to your corporate environment by applying some customization to the product. This is not required but should be thought of before starting a deployment in your organization.

At minimum, consider changing the „Corporate Registry Root" to your organization's name, because otherwise it will default to „Microsoft".

7.1 Setting the corporate registry key

This is a very important decision that has to be done in the planning phase. You should not change this setting after deployment since you would lose your previous inventory information. You might also prevent already queued packages from being executed. The corporate key specifies the registry sub-key under

```
HKEY_LOCAL_MACHINE\Software
```

This location is used by PKGShell to store all relevant information about queued and installed packages. If this key is not altered, it defaults to „Microsoft". If you customize it to „ACME", than the installed packages will be recorded under

```
HKEY_LOCAL_MACHINE\Software\ACME\Packages
```

So the queued packages will go to

```
HKEY_LOCAL_MACHINE\Software\ACME\PKGShell\Queue
```

If you have also customized the EXE name to „ACME_ISIS.EXE", it will be

```
HKEY_LOCAL_MACHINE\Software\ACME\ISIS\Queue
```

☞ On 64bit systems, these registry keys are automatically created as additional links under Software\Wow6432Node to allow identical access from 32bit apps under the WOW64 compatibility layer.

7.2 Choosing the EXE name

The name of the executable is PKGShell.exe by default. It is up to you to accept this default or change it to something that goes better with your corporate identity.

Reasons to go for a different name typically involve:

1. The executable must be used as a standard entry point to execute each and every package. It is crucial that all support personnel recognize this standard procedure to invoke a package.
2. Since this is a corporate standard it helps to use well known acronyms from within the organization.

Therefore many customers feel it is more suitable to choose a name like ACME_ISIS.EXE to emphasize that this is the corporate standard entry point. In this case „ACME" is the corporate acronym and „ISIS" stands for „integrated software installation service".

To change the name after downloading the PKGShell distribution ZIP:

1. Edit the provided .sms file and set exename= to your new name.

2. Rename all pkgshell.* files from the distribution to the name you desire.

The accompanying DLLs must always keep their names (e.g. PKGS_ENU.DLL).

Please also consider the following consequences:

1. If you choose a name with an underscore then the name of the service will be only the part *after* the underscore. E.g. if the exe is renamed to ACME_ISIS.EXE, then the service's name will be ISIS. Also keep this in mind in case you try to control the service directly via „net stop isis".
2. By changing the EXE's name you are also changing the default name of the expected package definition file: If the EXE's name is ACME_ISIS.EXE than the default file that is searched for would be ACME_ISIS.INI or ACME_ISIS.SMS. This name can still be overridden by explicit use of the command line parameter „-f".

7.3 Customizing the dialog logo

The dialog that is displayed when a new package arrives on the client has a default graphic logo on the left side:

The default logo uses a neutral image reminding of the windows setup

You may include text as a graphic element

Or any corporate color scheme

This can be easily customized by placing the logo as a JPG file on the client and using the parameter Logo= to point to the file.

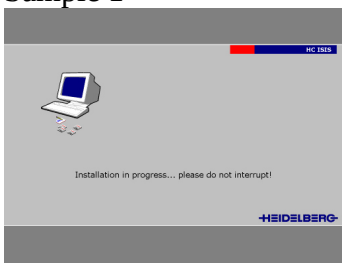
The image will be scaled as necessary to fit in the dialog window. For best results the dimensions of the image should be 149 pixels wide and at least 299 pixels high.

7.4 Customizing the installation background

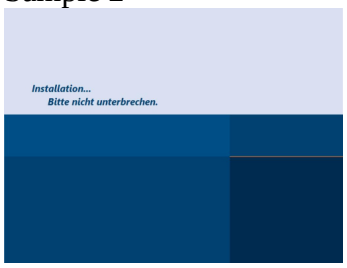
During the execution of the package, when the user is logged off, the background wallpaper is changed to indicate ongoing installation activity.

Default

Sample 1



Sample 2



To change the default background, just create a JPG file with the same name as the EXE and place it in your local %SystemRoot%.

For the default case (when you haven't renamed your EXE) this would be „%windir%\pkgshell.jpg".

7.5 Configuring and deploying service parameters

For a complete list of all possible service parameters please see section 8.2.2 on page 69.

The .CFG file can be used to import all required settings into the registry upon installation of the service. The CFG file has to have the same name as the EXE, but with the extension of CFG.

A configuration file for ACME_ISIS.EXE would thus be called ACME_ISIS.CFG:

```
[Parameters]
Blink = 1
Debug = 1
RegRoot = ACME
```

7.6 Extending the SMS inventory

If you are using Microsoft SMS or Configuration Manager it is highly recommended to extend the inventory in order to take advantage of the advanced PackageShell status reporting. This section explains how this is accomplished.

PackageShell writes all package status information in a structured format to the local client's registry. Since the SMS hardware inventory can easily report data from the registry this customization is a common approach. Technically, SMS uses WBEM and can only report WMI class instances. The built-in „WMI Registry Provider" comes in handy because it can be instructed to use a registry path as a source for WMI class instances. Two files need to be modified to make everything happen…

- ☞ When using the templates below, please remember to replace the ***marked*** strings with the respective names or abbreviations of your organization. This should be consistent with the customizations of the corporate registry root (section 7.1 on page 58).

7.6.1 Editing CONFIGURATION.MOF

We need to define a new class that will hold the package information. This is done by editing the „`Configuration.mof`" file on you site server, which is found under „`inboxes\clifiles.src\hinv`". Please append the following text:

```
#pragma namespace("\\\\.\\root\\cimv2")

[dynamic,provider("RegProv"),ClassContext("local|hkey_local_machine\\
Software\\Microsoft\\Packages")]
Class MS_Package:CIM_LogicalElement
{
```

```
    [Key]                                   string    Name;
    [PropertyContext("InstallDate")]        datetime  InstallDate;
    [PropertyContext("Status")]             string    Status;
    [PropertyContext("StatusDetail")]       string    StatusDetail;
    [PropertyContext("Build")]              uint32    Build;
    [PropertyContext("Contact")]            string    Contact;
    [PropertyContext("Description")]        string    Description;
    [PropertyContext("Duration")]           uint32    Duration;
    [PropertyContext("Signature")]          string    Signature;
    [PropertyContext("SMS")]                uint32    SMS;
    [PropertyContext("SourcePath")]         string    SourcePath;
    [PropertyContext("SourcePathOrigin")] string    SourcePathOrigin;
    [PropertyContext("SourceType")]         uint32    SourceType;
    [PropertyContext("User")]               string    User;
    [PropertyContext("AdvertID")]           string    AdvertID;
    [PropertyContext("PackageID")]          string    PackageID;
};
```

☞ In case you are using an older version of SMS, you may also need to enable the WMI Registry Provider („RegProv") earlier in the same file. The Registry Provider is enabled by default since ConfigMgr 2007.

7.6.2 Editing SMS_DEF.MOF

To activate the reporting of the new class as part of the hardware inventory, we need to edit the „SMS_DEF.MOF" file in the same directory of the site server. The following text should be appended near the end of the file:

```
#pragma namespace("\\\\.\\root\\cimv2\\sms")

[SMS_Report(TRUE),
SMS_Group_Name("Microsoft Packages"),
SMS_Class_ID("MS|Package|1.0") ]
Class MS_Package : SMS_Class_Template
{
    [SMS_Report(TRUE), Key]  string    Name;
    [SMS_Report(TRUE)]       uint32    Build;
    [SMS_Report(TRUE)]       string    Status;
    [SMS_Report(TRUE)]       string    StatusDetail;
    [SMS_Report(TRUE)]       datetime  InstallDate;
    [SMS_Report(FALSE)]      string    Description;
    [SMS_Report(FALSE)]      string    Contact;
    [SMS_Report(TRUE)]       string    Signature;
    [SMS_Report(TRUE)]       uint32    SMS;
    [SMS_Report(TRUE)]       string    SourcePath;
    [SMS_Report(TRUE)]       string    SourcePathOrigin;
    [SMS_Report(TRUE)]       uint32    SourceType;
    [SMS_Report(TRUE)]       string    User;
    [SMS_Report(TRUE)]       string    AdvertID;
    [SMS_Report(TRUE)]       string    PackageID;
    [SMS_Report(TRUE)]       uint32    Duration;
};
```

The inventory extension is automatically activated after the files are changed. It is good practice to try these modifications in a test environment before changing your production site. For more information, please refer to the Microsoft product documentation under the section „How to Extend Hardware Inventory”.

7.7 Creating SMS Reports

If you have extended the inventory to include your package data, you can access that information using the standard web-based reports.

7.7.1 Example usage of reports

A common request is to get an overview list of deployed packages with deployment counts:

Vista_WMF_Core-2_0	2312
Visual_Cpp_Redist_2005-8_0_59193	337
Visual_Cpp_Redist-2005_SP1	17031
Visual_Cpp_Redist-2008_SP1	17006
Visual_Cpp_Redist-2010	16

Clicking on one line gives a drilldown of the different states for a package:

Package	Build	State Name	Count
Visual_Cpp_Redist-2008_SP1	1	ABORTED	75
Visual_Cpp_Redist-2008_SP1	1	FAILED	5
Visual_Cpp_Redist-2008_SP1	1	OK	16922
Visual_Cpp_Redist-2008_SP1	1	WAITING	4

Clicking on the status gives a list of clients in the specific state:

Parameters:

Name of the package	Visual_Cpp_Redist-2008_SP1
Status of the package	FAILED
Build Number	1

27.07.2011 12:58:37 (Number of Records: 5)

Client	Build	Date	Detail
RQN03002	1	25.02.2011 12:45:00	1:Error 1935.An error occurred during the installation of assembly 'Microsoft.VC90.ATL,version=
RQN03357	1	26.02.2010 08:30:00	1:RETURN_ERROR#1603
RQD01542	1	12.10.2009 09:33:00	1:RETURN_ERROR#1603
RQN06781	1	12.10.2009 08:33:00	1:RETURN_ERROR#1603
RQD07012	1	08.10.2009 09:00:00	1:RETURN_ERROR#1602

These reports are relatively easy to build using the standard admin console. You can use the following SQL queries as a starting point…

7.7.2 View all deployed packages

```
SELECT
  pkg.Name0 as 'Package',
  count( distinct csp.IdentifyingNumber0 ) as 'Count'
FROM v_GS_MS_Packages0 pkg
  inner join v_R_System sys on pkg.ResourceID=sys.ResourceID
  inner join v_GS_Client0 csp on csp.ResourceID=sys.ResourceID
GROUP BY pkg.Name0
ORDER BY pkg.Name0
```

7.7.3 View all packages for a single client computer

```
SELECT
  pkg.Name0 as 'Package',
  pkg.Build0 as 'Build',
  pkg.InstallDate0 as 'Date',
  pkg.Status0 as 'Status',
  pkg.StatusDetail0 as 'Detail',
  pkg.Duration0 as 'Duration',
  pkg.User0 as 'User',
  pkg.PackageID0 as 'PKGID',
  adv.AdvertisementName as 'ADVID',
  pkg.SourcePath0 as 'Source Path',
  pkg.SourcePathOrigin0 as 'Origin'
FROM v_GS_MS_Packages0 pkg
  inner join v_R_System sys on pkg.ResourceID=sys.ResourceID
  left outer join v_Advertisement adv
    on pkg.AdvertID0=adv.AdvertisementID
WHERE sys.Name0=@Name
ORDER BY pkg.InstallDate0 desc
```

8 Reference

8.1 Package definition file parameters

8.1.1 Package Definition section

[Package Definition]	(required)
Name=	Name of Package, used as key for reporting (required)
Build=	Internal version number of package (required, integer)
Description=	Name and version as displayed to the user (required)
Contact=	Name of contact person responsible for package (optional)
Programs=	Comma-separated list of program section names, see below

8.1.2 Program section

[*Program*]	Name of program section (required, default is 'Install')
Command1=	First executable command (required)
CommandN=	Subsequent commands (optional)
Close=	Applications to close before execution (optional)
CopyLocal=	1 – Copy source files before queuing 0 – Run from network (default)
CopyLocalDir=	Destination of CopyLocal (optional, default is '%windir%\pkg\%name%')
DeleteLocalCopy=	1 – Purge local source after execution 0 – Leave it there (default)
Dialog=	1 – Display info and progress dialogs when Logoff=0 0 – Silently execute when Logoff=0 (default)
Direct=	1 – Run directly without queuing 0 – Queue and let service process package (default)
EstimatedRunTime=	Estimated time of execution (units default to minutes). If specified, a progress bar will be shown (optional, no default)
Express=	1 – Queue package before all other packages 0 – Queue as last package (default, first-in-first-out)
Inventory=	1 – Always trigger SMS hardware inventory (default), 0 – Don't trigger if package has Logoff=0
KeepLogDays=	Maximum age of log files to keep when re-executing the package (optional)
KeepLogFiles=	Maximum number of log files to keep when re-executing the package (optional)
Kickstart=	1 – Forcibly log user off without displaying dialog

	0 – Display dialog (default)
Last=	1 – Process package after all other packages 0 – Process after previously queued package (default)
Logoff=	1 – Require user to logoff, 0 – Start always (default)
MaxReject=	Number of times the user can postpone execution (optional) 0 – User can postpone endless times (default) N – User can postpone N times for 8 hours -1 – User cannot postpone execution at all
ManualClose=	1 – User needs to manually terminate applications, 0 – Applications are automatically closed (default)
NoCentralLog=	1 – Don't log this package status into the central log file, 0 – Log in central log (default)
NoRetry=	1 – Never retry, even if global service parameter Retry is set
NotAgain=	1 – Skip execution if a package with equal name and build number exists with status ok, 0 – Start always (default)
RunBefore=	A command that will be run before the package is queued (optional)
ResetHistory=	1 – Delete SMS execution history for this package, 0 – Don't modify (default)
Retry=	1 – Re-execute the package once if it runs into a status of ABORTED or FAILED.
Schedule=	1 – Schedule execution based on maintenance window, 0 – Start now (default)
Silent=	1 – No final message box if Kickstart=1 and Logoff=1, 0 – Display message box (default)
SuppressFinalReboot=	1 – Suppress pending reboot at end of package
Uninstall=	State that this package is a deinstallation (optional)

8.1.3 Command format

The basic and only required property of a command is its main value:

CommandN = <value>	The <value> is the command line that is executed by the operating system.

☞ Note that windows will automatically remove leading and terminating double quotation marks from the value if present. This can be a problem if both your command and your parameter are using double quotes:

Problem	`CommandN = "my prog.exe" "my param"`	→ `my prog.exe" "my param`
Better	`CommandN = '"my prog.exe" "my param"'`	→ `"my prog.exe" "my param"`

This is not a bug in PKGShell but a Windows feature you should be aware of.

A command is ***external*** by default because it is interpreted and executed by the operating system. There are only a few exceptions where the value of a command is treated as ***internal*** and is processed by PKGShell:

EXIT	The execution of the package will terminate here.
GOTO <label>	Goto command <label> or command number.
IF : <test> : <number>	Goto command <number> if <test> is true. <number> can be an integer or a label.
PKG:<name>[:<program>]	Execute package <name> as a sub-package.
RUNTASK <name>	Start and wait for a Windows scheduled task.
SET <name> = <value>	Define or modify the value of a variable.
SUB:<name>	Execute commands of section [SUB:<name>]
TEST:<name>	Perform the tests defined in section [TEST:<name>]
WAIT <duration>	Wait for the specified duration in seconds. Can also use “1.5 hours” or “30 min”. No duration means forever. Combine with .While or .Success to specify stop criteria.

8.1.4 Command properties

A command can be given optional properties (attributes) that are used by PKGShell to change the default behavior of a command. These properties are used by appending a dot and the properties name to the command:

CommandN .CD=	The working directory to use when executing the command. If not set, this will default to '%SourcePath%'
CommandN .DoReboot=	1 – Always reboot after command (optional) 0 – Reboot only if the command requests so
CommandN .Foreach= .Foreach:<Name>=	List of values to iterate. The command will be executed repeatedly for each given value. The current value can be used within the command by using the default variable %_%. Optionally you can use a different variable by specifying a <Name>.
CommandN .Hidden=	1 – Hide command window during execution 0 – Display on active desktop (default)
CommandN .IgnoreCopyLocal=	1 – Don’t copy this command’s sub package 0 – Copy sub package if CopyLocal=1 (default)

CommandN .IgnoreError=	1 – Ignore return code (optional) 0 – Report error on non-zero return code
CommandN .Label=	A string that can be used as a jump destination by another GOTO or IF command (optional)
CommandN .LogFile=	Path to a log file who's content should be displayed on the progress dialog (optional, defaults to package log)
CommandN .NoExpand=	1 – do not expand variables on the command line 0 – expand variables (default)
CommandN .NoNetworkWait=	1 – don't wait for workstation service to come up 0 – wait for network before starting command (default)
CommandN .NoWait=	1 – Continue with next command before this is finished 0 – Wait for command to terminate (default)
CommandN .Required=	Test expression. If it evaluates to FALSE, the command execution is skipped (optional)
CommandN .Retry=	1 – Perform a retry after reboot of this command on failure 0 – Perform a retry of the package if global Retry=1
CommandN .Success=	Test expression. If it evaluates to FALSE, the command's result is considered FAILED (optional)
CommandN .SuccessCodes=	A list of numbers that will be treated as successful return codes. Hex numbers must start with 0x (optional)
CommandN .Session0=	1 – Execute command in the default service session '0' 0 – Run in visible session (default)
CommandN .SuppressReboot=	1 – Reboot request will be ignored (optional) 0 – Reboot if MSI return code requests so
CommandN .Timeout=	n – Wait for maximum of n seconds for completion 0 – Wait indefinitely (default)
CommandN .TolerateReboot=	1 – Tolerate if command performs a reboot (optional) 0 – Report error on reboot
CommandN .Until=	The command is repeated until the given test expression is true
CommandN .While=	The command is repeated while the given test expression is true

8.2 PKGShell.exe command line parameters

8.2.1 For daily work

/f	**Path to package definition file** If this is not specified while queuing a package, the current directory is used.

/k	**'Kickstart'** Starts the execution of all queued packages immediately as if the user had accepted the dialog box. The user is logged off and no dialog is shown.
/l	**'List queued packages'** List the locally queued packages and their status. This is practical for troubleshooting clients from the command line.
/o	**'Override Required/PreQueue tests'** Ignore what is defined under TEST:Required and TEST:Prequeue and execute the package as if the tests were true.
/p	**'PreQueue test only'** Perform whatever is defined under TEST:Required and TEST:Prequeue. If the test returns ok, return 0. Otherwise, return 800 or 801.
/q	**'Queue Only'** Submit a package to the queue without starting the PKGShell service. Use this to queue several packages at once before starting execution. Using this option will also ignore the test sections 'Required' and 'PreQueue'
/x [*]	**Delete packages in queue** Purges some or all packages from the local queue. Wildcards for package names are accepted. The deleted package's status is set to '`ABORTED:DELETED`' for active packages or to '`CANCELED:DELETED`' for waiting packages. The service is stopped.

8.2.2 For PKGShell service maintenance

/delete	**Deinstall the PKGShell service** Remove the PKGShell service from the system so it does not start every time the computer is booted.
/i	**Install PKGShell service** Installs or updates the existing service without queuing a package. The installation is normally performed automatically whenever a package is queued.
/unlink	**Remove registry links** Cleans up the 32bit registry links created by the PKGShell service installation. This is not automatically done by the /delete parameter and has to be called explicitly to completely clean a machine.

8.3 PKGShell service parameters

The following parameters can be used to change the PKGShell service's default behavior. The parameters have to exist as registry values of type REG_SZ under the registry key:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\PKGShell\Parameters

All parameters can be easily customized and are also best deployed using a CFG file as described in section 7.5 on page 61. The complete List of available parameters is:

Value	***Default***	***Explanation***
AutoAcceptMin	10	Number of minutes after which a package will be auto-accepted if no user is logged on.
Beep	0	If set to 1, a beeping sound is emitted every two seconds during execution.
Blink	0	If set to 1, a light is flashed every two seconds during execution.
CopyLocalPath	%windir%\PKG	Path under which packages are copied if parameter CopyLocal=1 is used.
Debug	0 (Minimal number of eventlog entries)	If set to a higher value (up to 5), this will increase the number and verbosity of PKGShell generated eventlog.
DialogPercent	0 (Hardcoded width 640)	If set to number less or equal to 100 the dialog box width is sized relatively to the dimensions of the desktop.
DialogTimeout	8	Hours after which the rejected dialog box is redisplayed.
InventoryRun	(not set)	If this key is set, the value is executed as a command each time an inventory needs to be triggered.
LegacyStatus	(not set)	When not defined, all intermediate states are written to the registry under Execution. Only when a package reaches OK, the data is copied to the Packages key. If set to 1, the old behavior is used: The Execution key is not used at all and the Packages reg key gets all states.
License	(not set)	String obtained by purchasing a license
LogFile	%name%.log %name-program%.log	File name of package log file. Includes program if this is not "install".
Logo	(not set)	Path to external logo bitmap for

		package info dialog
LogPath	%windir%\Logs	Directory under which all log files are stored
LogSub	0	If set to 1, sub packages (PKG:) will log into their separate log files.
MaintenanceWindow	SAT-1200-0100 (Saturday 12p.m. for duration of 1 hour)	Installations started with Schedule=1 will start at the time given.
OSDQueue	0	If set to 1, packages will be 'queued only' while a task sequence is in progress (as with -q). The queue will be started later using SMSTSPostAction.
RegRoot	Microsoft	Key under HKLM\Software that contains subkeys 'Queue', 'Execution' and 'Packages'
RegExec	pkgshell\Execution	Registry execution report path
RegQueue	pkgshell\Queue	Registry queue path
RegPkg	Packages	Registry package report path
RemoteLog		If set to a UNC path all status changes will be written into this file as well.
ResetHistory	0	If set to 1 will by default delete execution history of other programs for the same package.
Retry	0	If set to 1, packages with status ABORTED or FAILED are re-executed once.
ShowFinalDialog	0	If set to 1, the final dialog is always shown, even if no user was logged on.
SMSPause	1	If set to 0, the SMS/ConfigMgr Client service is not paused during execution.
StatusMigration	0	If set to 1, the Packages reg key is an exact mirror of the Execution reg key.
TagActiveSetup	0	If set to 1, will create a value named PKGShell under each new ActiveSetup entry. If set to string, will use this string as value name.

8.4 Obsolete package definition file parameters

Since 1997 a lot of packages were authored using PKGShell. It is a top priority to maintain backwards compatibility and thus to never break an existing package in production. Thus, if a name of a parameter is changed, the old version will remain functional.

The following historical package parameters are working but deprecated:

FinalStatus= (deprecated)	Ignore all (!) command return codes and report this string as the final status instead of 'OK' (optional)
IgnoreError= (see .IgnoreError)	Comma-seperated list of command numbers of which the return code will be ignored (optional)
IgnoreReboot= (see .TolerateReboot)	Comma-separated list of command numbers which may perform reboot by themselves (optional)
Reboot= (see .DoReboot)	Comma-seperated list of command numbers after which a reboot will be performed (optional)
SuppressReboot= (see .SuppressReboot)	Comma-seperated list of command numbers for which a reboot request will be ignored (optional)
Verify= (see [TEST:Success])	Path of file whose existence will be verified after execution (optional)