Oracle® Solaris 11 Express Distribution Constructor Guide

Part No: 820–6564–12
November 2010

ORACLE

110425@25097

Contents

◆ ◆ ◆ CHAPTER 1

Introduction to the Distribution Constructor

You can use the distribution constructor to build custom Oracle Solaris images. Then, you can use the images to install the Oracle Solaris software on individual systems or multiple systems. You can, also, use the distribution constructor to create Virtual Machine (VM) images that run the Oracle Solaris operating system.

- If you have not used the distribution constructor before, read the following introduction.
- If you are ready to use the distribution constructor, go to Chapter 2, "Design and Build Oracle Solaris Images."
- If you want to build a Virtual Machine, see the instructions at Chapter 3, "x86: Design and Build a Virtual Machine."

What is the Distribution Constructor?

The distribution constructor is a command-line tool for building preconfigured Oracle Solaris images and Virtual Machine images. The tool takes an XML manifest file as input, and builds an ISO image or Virtual Machine image that is based on the parameters specified in the manifest file. An ISO image is an archive file, also known as a disc image, of an optical disc in a format defined by the International Organization for Standardization (ISO).

Optionally, a USB image can be created, based on a generated x86 ISO image. Depending on the image configuration, these ISO or USB images can be bootable. They can be installed on a system or run in a live media environment. An ISO image can be burned to a CD or DVD. A USB image can be copied to a flash drive.

Note – The Distribution Constructor creates a USB image that could work in various types of flash memory devices, if those devices have driver support provided by the Oracle Solaris release. The `usbcopy` utility is the only utility that can be used to copy the USB image into a USB flash drive. This `usbcopy` utility is available with the Oracle Solaris release.

What Kind of Oracle Solaris Images Can I Build?

You can use the distribution constructor to create the following types of Oracle Solaris images.

- **Oracle Solaris x86 live CD image** – Using the distribution constructor, you can create an x86 ISO image, called a live cd or "slim CD" image, that is comparable to the live CD image that's distributed with the Oracle Solaris release. This live CD image boots to a GNOME desktop, where you can explore the Oracle Solaris operating system. From this desktop, you have the option to install the operating system. You can use the distribution constructor to modify the content of this live CD ISO image by adding or removing packages. You can revise the default settings for the resulting booted environment to create a custom ISO image or USB image. See Chapter 2, "Design and Build Oracle Solaris Images."

  For more information about using the Oracle Solaris x86 live CD, see *Getting Started With Oracle Solaris 11 Express* .

- **x86 or SPARC Oracle Solaris text installer image** – Using the distribution constructor, you can create an ISO image that can be used to initiate a text installation of the Oracle Solaris operating system for x86 or SPARC machines. This text installer image can be used to install the Oracle Solaris operating system on machines that do not have graphics cards. The text installer does not install a desktop with the operating system. But, you can add packages to the operating system, including packages for a GNOME desktop, after the installation.

  For more information about performing a text installation, see *Getting Started With Oracle Solaris 11 Express* .

- **x86 or SPARC ISO Image for Automated Installations** – The Oracle Solaris release includes the automated installer tool. The automated installer (AI) is used to automate the installation of the Oracle Solaris OS on one or more SPARC and x86 systems over a network. The installations can differ in architecture, packages installed, disk capacity, and other parameters. The automated installer uses a SPARC or x86 AI ISO image to install the Oracle Solaris OS to client systems. You can use the distribution constructor to create a SPARC AI ISO image that can be used to install the Oracle Solaris OS on SPARC clients, or to create an x86 AI ISO image that can be used to install the Oracle Solaris OS on x86 clients.

  For information about using the automated installer, see the *Oracle Solaris 11 Express Automated Installer Guide* .

  Note – When using the distribution constructor, you can only create SPARC images on a SPARC system. And, you can only create x86 images on an x86 system. Also, the Oracle Solaris release version on your system must be the same as the release version of the AI images that you use with the distribution constructor.

- **x86 Oracle Solaris Virtual Machine** – You can use the distribution constructor to create a Virtual Machine that's preinstalled with the Oracle Solaris operating system. See Chapter 3, "x86: Design and Build a Virtual Machine."

What are the Components in the Distribution Constructor?

The distribution constructor uses a few core components to design and build images.

- "Manifest Files" on page 7
- "Finalizer Scripts" on page 7
- "`distro_const` Utility" on page 8 with "Checkpointing Options" on page 8

See the following short descriptions of these core components.

Note – For complete instructions about how to use the distribution constructor, see Chapter 2, "Design and Build Oracle Solaris Images."

Manifest Files

The distribution constructor creates images based on settings specified in XML files, called *manifest files*. The manifest files contain specifications for the contents and parameters of the ISO images that you create using the distribution constructor. The distribution constructor contains default manifests that can be used to create a custom live CD, an x86 or SPARC text installer image, an x86 or SPARC AI ISO image, or a Virtual Machine image.

All the fields in each manifest file provide preset, default values that will create the type of image you need. Optionally, you can manually edit these preset fields in a manifest file to further customize the resulting image.

For instructions about how to customize manifest files, see "Customize Your Image by Editing the Manifests" on page 10.

Finalizer Scripts

The distribution constructor enables you to specify scripts that can be used to make installation customizations based on the type of image you are building. These scripts are called *finalizer scripts*. The manifest files point to the finalizer scripts, and the finalizer scripts transform the generic image into a media-specific distribution. The default scripts are provided when you install the distribution constructor package.

For instructions about how to create and use finalizer scripts, see "Further Customize an Image Using Finalizer Scripts" on page 12.

`distro_const` Utility

The distribution constructor package also includes a command-line utility, the `distro_const` command, that actually builds your image for you. After you have finished editing the image blueprint in a manifest file to suit your requirements, you run the `distro_const` command to build your image.

For instructions about how to use the `distro_const` command, see "Building an Image" on page 17.

Checkpointing Options

You can use the options provided in the `distro_const` command to stop and restart the build process at various stages in the image-generation process, in order to check and debug the image that is being built. This process of stopping and restarting during the build process is called *checkpointing*.

For instructions about how to use checkpointing, see "Building an Image in Stages" on page 18.

◆ ◆ ◆ CHAPTER 2

Design and Build Oracle Solaris Images

To design and build Oracle Solaris images with the distribution constructor, complete the following four tasks.

Note – Building a Virtual Machine has unique instructions, different from the general instructions below. If you want to build a Virtual Machine, see Chapter 3, "x86: Design and Build a Virtual Machine."

1. Review the "System Requirements" on page 9, and install the distribution constructor application on your system.
2. Optional: "Customize Your Image by Editing the Manifests" on page 10.
3. Optional: "Further Customize an Image Using Finalizer Scripts" on page 12.
4. "Building an Image" on page 17.

For complete instructions, see the following sections.

System Requirements

In order to use the distribution constructor, set up the following on your system.

TABLE 2–1 System Requirements

Requirement	Description
Disk space	Confirm that you have sufficient space on your system to use the distribution constructor. The recommended minimum size for your distribution constructor work space is 8 Gbytes.

TABLE 2–1 System Requirements *(Continued)*

Requirement	Description
Oracle Solaris release	You must have the SPARC or x86 Oracle Solaris operating system (OS) installed on your system. Note the following considerations. ■ Your installed Oracle Solaris system must have network access. The distribution constructor accesses Image Packaging System (IPS) repositories that are available on the network to retrieve packages for the ISO image. You must have network access to the repositories that you specify in the manifest file. ■ When using the distribution constructor, you can only create SPARC images on a SPARC system. And, you can only create x86 images on an x86 system. Also, the Oracle Solaris release version on your system must be the same as the release version of the AI images that you use with the distribution constructor. **Note –** To run the distribution constructor on your system, you will need to assume the root role by executing the `su -` command.
Required packages	Install the `distribution-constructor` package, which contains the distribution constructor application, on your system. You can use the Package Manager tool to install the required package. The Package Manager is available as an icon on the desktop of the Oracle Solaris operating system and on the menu bar on the desktop. On the menu bar, go to System>Administration>Package Manager. Alternately, use IPS commands such as the following to install this package: `# pkg install distribution-constructor`

Customize Your Image by Editing the Manifests

The distribution constructor creates images based on settings specified in XML files, called *manifest files*. The manifest files contain specifications for the contents and parameters of the ISO images that you create using the distribution constructor. The distribution constructor contains default manifests that can be used to create a custom live CD, an x86 or SPARC text installer image, an x86 or SPARC AI ISO image, or a Virtual Machine.

Manifests specify parameters such as the following:

- Names of the packages to be included in the image
- Network location of the repository to access to retrieve packages for the image
- Name and location of scripts used to finalize the creation of the new image

Tip – When you use the distribution constructor to create ISO images, note the following:

The root archive for x86 images differs from the root archive for SPARC images. The whole root archive, or `boot_archive`, for x86 images is a UFS filesystem, compressed by using `gzip`. The SPARC platform does not support the compression of the whole root archive in this way. Instead, SPARC root archives use DCFS, which compresses each file individually. These individually compressed files might require specific handling in the manifest. For instructions, see the `<boot_archive_contents>` field in the `dc_manifest`(4) man page.

The default manifest files included in the `distribution-constructor` package are listed in the following table.

TABLE 2–2 Default Manifests

Manifest Type	Manifest Location	Description
x86 live CD ISO image	`/usr/share/distro_const/slim_cd/all_lang_slim_cd_x86.xml`	This manifest is used to create an ISO image comparable to the Oracle Solaris live CD, also called the "slim CD."
x86 text installer ISO image	`/usr/share/distro_const/text_install/text_mode_x86.xml`	This manifest is used to create an ISO image that you can boot to initiate a text installation of the Oracle Solaris OS on x86 machines.
SPARC text installer ISO image	`/usr/share/distro_const/text_install/text_mode_sparc.xml`	This manifest is used to create an ISO image that you can boot to initiate a text installation of the Oracle Solaris OS on SPARC machines.
SPARC AI ISO image	`/usr/share/distro_const/auto_install/ai_sparc_image.xml`	This manifest is used to create a SPARC AI ISO image for automated installations of the Oracle Solaris OS to SPARC clients.
x86 AI ISO image	`/usr/share/distro_const/auto_install/ai_x86_image.xml`	This manifest is used to create an x86 AI ISO image for automated installations of the Oracle Solaris OS to x86 clients.
x86 Virtual Machine	`/usr/share/distro_const/vmc/vmc_image.xml`	This manifest is used to create a Virtual Machine image. For instructions, see Chapter 3, "x86: Design and Build a Virtual Machine."

Modifying Manifest Contents

All the fields in each manifest file provide preset, default values that will create the type of ISO image that you need, as well as, for x86 systems, USB images. You have the option to manually edit these preset fields in a manifest file to further customize the resulting image.

If you want to modify the manifest information, use the following process:

1. Copy one of the default manifests and create a custom manifest file with a new file name.

   You will reference the manifest file by name when you use the `distro_const` command to create an image.

   Note – Always back up the original manifest file and the default scripts before copying them.

2. Edit the manifest fields to suit your needs. See the `dc_manifest`(4) man page for instructions.
3. If the default finalizer scripts do not leave the `build_area` as you need it, you can create your own scripts to make further modifications. If you do create new scripts, update the script references in the finalizer section of the manifest file.

   For instructions about editing the finalizer script section in the manifest file, see the `dc_manifest`(4) man page.

   For instructions about creating new scripts, see "Further Customize an Image Using Finalizer Scripts" on page 12.
4. When you have completed any revisions to the manifest file and, optionally, customized the finalizer scripts as described in the next section, you can proceed with running the `distro_const` utility to create an image. For instructions, see "Building an Image" on page 17.

Further Customize an Image Using Finalizer Scripts

The distribution constructor enables you to specify scripts that can be used to make installation customizations based on the type of image you are building. These scripts are called *finalizer scripts*. The manifest files point to the finalizer scripts, and the finalizer scripts transform the generic image into a media-specific distribution. A set of default scripts are provided in the application packages.

The distribution constructor application includes default scripts in the `/usr/share/distro_const` directory and subdirectories. These scripts are referenced in the finalizer section of the manifest files.

For example, some of the default finalizer scripts that are used to create an x86 live CD, or "slim CD," are listed in the following table.

TABLE 2–3 x86 Live CD Finalizer Scripts

Finalizer Script	Description
`pre_boot_archive_pkg_image_mod`	Modifies image area for all types of images
`slim_cd/slimcd_pre_boot_archive_pkg_image_mod`	Modifies image area, specifically for live CD image
`slim_cd/slimcd_gen_cd_content`	Generates the list of files that are part of the live CD image
`boot_archive_initialize.py`	Initializes the boot archive
`slim_cd/slimcd_boot_archive_configure`	Configures the boot archive, specifically for the live CD image
`boot_archive_configure`	Configures the boot archive
`boot_archive_archive.py`	Archives the boot archive
`slim_cd/slimcd_post_boot_archive_pkg_image_mod`	Makes post-build modifications to the boot archive image area, specifically for live CD image
`grub_setup`	Initializes the Grub menu
`post_boot_archive_pkg_image_mod`	Makes post-build modifications to the boot archive image area
`create_iso`	Creates an ISO image
`create_usb`	Creates a USB image

Creating Custom Finalizer Scripts

It is recommended that you use the supplied finalizer scripts without editing them. You do, however, have the option to write and add your own scripts to perform additional operations as described below. If you do create new scripts, edit the manifest files to point to these new scripts.

Note – Support for scripts is limited to the unmodified, default scripts that are supplied with the application packages. If you choose to customize these scripts, back up the original scripts first.

1. Plan your new scripts. Use the existing scripts as models for new scripts. Review the "Characteristics of Finalizer Scripts" on page 14.

   See, also, the following sample custom scripts:

   - "Sample Script for Adding Packages to an Image" on page 35
   - "Sample Script for Testing Alternate Packages" on page 39
   - "Sample Script for Adding an SVR4 Package to an Image" on page 41

2. Create your new scripts.

3. Add your new scripts to either the `/usr/local/` directory, your home directory, or elsewhere on the system or network. Make sure that the root role can execute these scripts.
4. Add the new script names in the finalizer section of the appropriate manifest file. Be sure to specify the full path to your scripts in the manifest file, even if they are in the `/usr/share/distro_const` directory.
5. When you add a reference for a new script in the finalizer section of a manifest file, you must specify a checkpoint name that is used to pause the image build before or after this script performs its task. Optionally, you can include a custom message associated with the checkpoint name. If this message is omitted, the path of the script is used as the default checkpoint message.

Note – Use meaningful names for checkpoint names instead of using numbers. If new scripts are added, the new steps for those new scripts will disrupt a numbered checkpoint order.

For example, the following reference in a manifest file specifies the checkpoint name of "ba-arch" for a boot archive archiving script, and the associated message is "Boot archive archiving."

```
<script name="/usr/share/distro_const/boot_archive_archive.py">
<checkpoint name="ba-arch" message="Boot archive archiving"/>
</script>
```

This example allows you to specify a `distro_const` command to build your image, pausing or resuming from the "ba-arch" checkpoint. The build will pause just before the boot archive archiving script performs its task.

For instructions about how to refer to checkpoints in the `distro_const` command, see "Building an Image in Stages" on page 18.

Characteristics of Finalizer Scripts

When you create your own finalizer scripts, note the following:

- Scripts can be Python programs, shell scripts, or binaries.
- Scripts are executed in the order that they are listed in the manifest file.
- Standard output (`stdout`) and error output (`stderr`) of commands executed within finalizer scripts (both shell and python modules) are captured in the log files. `stderr` is captured in the simple log. Both `stdout` and `stderr` are captured in the detail log.
- The distribution constructor passes five initial arguments to all scripts that are executed. These five arguments are not included as entries in the manifest file. The manifest file specifies additional arguments passed in after these five arguments. The initial arguments are as follows.

TABLE 2–4 Finalizer Script Arguments

Argument	Description
Server socket file name	The first argument is the manifest reader socket. This argument specifies the socket that is used with `/usr/bin/ManifestRead` for accessing the manifest data. See the section about "Using the Manifest Reader" on page 16.

Package image area path	The second argument is the `PKG_IMG_PATH`, which specifies the path to the area where the package image is created. Use this argument to locate a file in the package image area. The following example checks whether the user, "jack," is in the package image area's password file.
	`PKG_IMG_PATH=$2`
	`/usr/bin/grep jack $PKG_IMG_PATH/etc/passwd >/dev/null`
	`if [[ $? == "0" ]] ; then`
	`  print "Found Jack"`
	`fi`

Temp directory	The third argument specifies a directory that is used when creating temporary files needed during the build process. In the following example, you create a file in the temporary directory for building the boot archive.
	`TMP_DIR=$3`
	`/usr/sbin/mkfile $TMP_DIR/boot_archive_archive`
	`/usr/sbin/lofiadm -a $TMP_DIR/boot_archive_archive`

Boot archive build area	The fourth argument is the boot archive build area, where the boot archive files are gathered. See the following example from the `boot_archive_configure` module, which adds the file, `/etc/nodename`, to the boot archive. This file gives the system the hostname, "solaris."
	`BR_BUILD=$4  # Boot archive build area`
	
	`# Set nodename to solaris`
	`echo "solaris" > $BR_BUILD/etc/nodename`

Media area	The fifth argument specifies where the finished media is deposited. In the following example, the `create_iso` script uses this argument to place the resultant ISO image.
	`MEDIA_DIR=$5`
	`...`
	`DIST_ISO=${MEDIA_DIR}/${DISTRO_NAME}.iso`
	`...`

TABLE 2–4 Finalizer Script Arguments *(Continued)*

Argument	Description

Additional arguments	A list of additional arguments to be passed into a script is tagged in the manifest with the `<argslist>` tag. The first of these arguments is passed in as `arg6`. Enclose each list item in double-quotes. When no double-quotes are used, or if one set of double-quotes encloses the entire string, the entire string including spaces and newlines is interpreted as one argument. Do not use commas between arguments.
	
	In the following example from the `slim_cd_x86.xml` manifest file, two additional arguments are passed to the `boot_archive_configure` script, as `arg6` and `arg7`:
	
	`<argslist>`
	`"/usr/share/distro_const/slim_cd/slimcd_generic_live.xml"`
	`".livecd"`
	`</argslist>`
	
	Another method for specifying additional arguments is to use key-value pairs. See the following section.

Using the Manifest Reader

The distribution constructor passes the manifest reader socket as the first argument into finalizer scripts. In a finalizer shell script, pass this argument as the first argument to `/usr/bin/ManifestRead` for accessing the manifest data. In a python module, pass this argument to instantiate a ManifestRead() object.

The following Shell script example calls `ManifestRead` to request the `name` item from the manifest file. `ManifestRead` returns zero or more items, each on its own line. If `ManifestRead` is given multiple items to search for, the lines returned contain both the item being searched for and a result.

```
MFEST_SOCKET=$1

VOLNAME=‘/usr/bin/ManifestRead ${MFEST_SOCKET} "name"‘
if [ "XX${VOLNAME}" == "XX" ] ; then
       print -u2 "$0: Error retrieving volume ID"
       exit 1
fi
```

The following example shows how to make use of `ManifestRead` from a Python script:

```
from osol_install.ManifestRead import ManifestRead

# get the manifest reader object from the socket
manifest_reader_obj = ManifestRead(MFEST_SOCKET)

# get boot_archive compression type

BR_COMPR_TYPE = get_manifest_value(manifest_reader_obj,
   "img_params/output_image/boot_archive/compression/type")
if (BR_COMPR_TYPE == None):
       raise Exception, (sys.argv[0] +
           ": boot_archive compression type missing from manifest")
```

Specifying Key-Value Pairs

Another method for passing arguments into scripts is to specify a key-value pair. This method is useful for passing the same argument into multiple scripts without duplication. A script can access a keyed value by specifying the key to `/usr/bin/ManifestRead` from within the script. Provide the server socket as the first argument and then provide the node paths to the items whose values are needed, as in the following examples.

EXAMPLE 2–1 Shell Script

The following example calls `ManifestRead` from a shell script to obtain a keyed value.

```
...
  MFEST_SOCKET=$1
  ...
  /usr/bin/ManifestRead -k $MFEST_SOCKET iso_sort
  iso_sort_file=‘/usr/bin/ManifestRead $MFEST_SOCKET iso_sort‘
```

EXAMPLE 2–2 Python Script

The following example calls `ManifestRead` from Python to obtain the same keyed value.

```
from osol_install.ManifestRead import ManifestRead

...
  IS_KEY = True

  iso_sort_file = manifest_reader_obj.get_values("iso_sort", IS_KEY)
  fd = open(iso_sort_file,....)
```

Building an Image

After you have set up the manifest file that you plan to use and, if desired, customized the finalizer scripts, you are ready to build an image by running the `distro_const` command.

Note – To run the `distro_const` command, assume the root role by executing the `su -` command.

You can use the `distro_const` command to do either of the following:

- Build an image in one step.
- Build an image, but pause and restart the build as needed to examine the content of the image and debug the finalizer scripts during the build process.

The full syntax for this command is as follows:

`Syntax: distro_const build [-R] [-r` *step*`] [-p` *step*`] [-l]` *manifest*

Building an Image in One Step

To run a complete build of an image without pausing, use the basic `distro_const` command without options as follows:

```
# distro_const build manifest
```

Replace *manifest* with the name of the manifest file to be used as the blueprint for your image. The `build` subcommand is required. When you type this command, the distribution constructor pulls the needed packages for the image and builds the image to the specifications that you set up in the manifest file.

Building an Image in Stages

You can use the options provided in the `distro_const` command to stop and restart the build process at various stages in the image-generation process, in order to check and debug your selection of files, packages, and scripts for the image that is being built. This process of stopping and restarting during the build process is called *checkpointing*.

Checkpointing supports the process of developing and debugging images. You can start building an image, pause at any stage where you want to stop and examine the contents of the image, then resume building the image. Checkpointing is optional.

Note – The checkpointing feature is enabled by default in the manifest file. But a ZFS dataset, or a mount point that correlates to a ZFS dataset, must be specified as the build area.

Alternately, you can disable checkpointing in the manifest file by setting the `checkpoint_enable` parameter to `false`.

Use the checkpointing options that are available in the `distro_const` command as described in the following basic instructions. See, also, the `distro_const`(1M) man page.

▼ Building an Image in Stages by Using Checkpoint Options

1 Before you build the image, check the valid steps at which you can choose to pause or resume the build by using the following command:

```
# distro_const build -l manifest
```

Note – The `build` subcommand is required.

A checkpoint in the build process occurs after each finalizer script is executed.

This command displays the valid checkpoints at which you can pause or resume building an image. Use the step names provided by this command as valid values for the other checkpointing command options.

For example, the following command confirms which checkpoints are available, given a manifest file named *slim_cd_x86.xml*.

```
# distro_const build -l slim_cd_x86.xml
```

After the command is run, the valid checkpointing steps are displayed. Checkpointing steps can include the following.

```
Step           Resumable Description
-------------- --------- -------------
im-pop              X    Populate the image with packages
im-mod              X    Image area modifications
slim-im-mod              Slim CD image area modifications
ba-init                  Boot archive initialization
slim-ba-config           Slim CD boot archive configuration
ba-config                Boot archive configuration
ba-arch                  Boot archive archiving (64-bit)
ba-arch-32               Boot archive archiving (32-bit)
slim-post-mod            Slim CD post boot archive image area modifications
grub-setup               Grub menu setup
post-mod                 Post boot archive image area modification
gen-slim-cont            Generate Slim CD image content list
iso                      ISO image creation
usb                      USB image creation
```

Note – In the command output, a check in the resumable field indicates that you can restart the build from this step.

2 Use the following command to build an image, and to pause building the image at the specified step.

```
# distro_const build -p step manifest
```

Note – The `build` subcommand is required. The *step* and *manifest* fields are required.

For example, the following command starts building an image and pauses the build before step *im-mod* modifies the image area:

```
# distro_const build -p im-mod slim_cd_x86.xml
```

3 Resume the build, either from the last step executed or from a specified step, by using one of the following alternatives:

- **Use the following command to resume building the image from a specified step.**

  ```
  # distro_const build -r step manifest
  ```

Note – The specified step must be either the step at which the previous build stopped executing, or an earlier step. A later step is not valid. The *step* and *manifest* fields are required. The `build` subcommand is required.

For example, the following command resumes building the image where the *im-mod* modifies the image area:

```
# distro_const build -r im-mod slim_cd_x86.xml
```

- **Use the following command to resume building the image from the last step executed.**

Note – The *manifest* argument and the `build` subcommand are required.

```
# distro_const build -R manifest
```

For example, the following command resumes building the image from wherever the build had paused.

```
# distro_const build -R slim_cd_x86.xml
```

Troubleshooting the Distribution Constructor

Review the following troubleshooting items.

I Get Error Messages When I Download a Package

Make sure the `pkg(1)` command on your system is working correctly, and your connection with the IPS server is stable. Sometimes, IPS times out when trying to download a big cluster of packages. To check outside of the distribution constructor environment, try to mimic what the constructor does in terms of installing packages. Assume the root role by executing the `su -` command. Then, try the following commands, confirming that the commands work correctly.

```
pkg image-create -F -a opensolaris.org=http://pkg.opensolaris.org
pkg -R /tmp/test_img install SUNWcsd
pkg -R /tmp/test_ima install SUNWcs
pkg -R /tmp/test_img install slim_install
```

How to Debug Manifests With Validation Errors

If a manifest does not validate, as could be the case after the manifest has been changed, run the `ManifestServ` utility in a verbose mode to find the error.

The `ManifestServ` utility, `/usr/bin/ManifestServ`, with no arguments displays the following usage:

```
ManifestServ
Usage: /bin/ManifestServ [-d] [-h|-?] [-s] [-t] [-v] [-f <validation_file_base> ]
    [-o <out_manifest.xml file> ] <manifest.xml file>
where:
  -d: turn on socket debug output (valid when -s also specified)
  -f <validation_file_base>: give basename for schema and defval files
      Defaults to basename of manifest (name less .xml suffix) when not provided
  -h or -?: print this message
  -o <out_manifest.xml file>: write resulting XML after defaults and
      validation processing
  -t: save temporary file
      Temp file is "/tmp/<manifest_basename>_temp_<pid>
  -v: verbose defaults/validation output
  -s: start socket server for use by ManifestRead
```

The distribution constructor validates the manifest against an XML schema and a default manifest. `ManifestServ` enables you to perform a manual validation, using a verbose mode which shows where any problems are.

EXAMPLE 2–3 Debugging Schema Validation Errors

The following example demonstrates a case where the manifest didn't validate against the schema. The boldface message below indicates this is a schema validation error.

```
# distro_const build my_distro.xml
/usr/share/distro_const/DC-manifest.defval.xml validates
/tmp/all_lang_slim_cd_x86_temp_7861.xml:350: element pair:
Relax-NG validity error : Element pair failed to validate attributes
/tmp/my_distro_temp_7861.xml fails to validate
validate_vs_schema: Validator terminated with status 3
validate_vs_schema: Validator terminated abnormally
Error validating manifest against schema
/usr/share/distro_const/DC-manifest.rng

#
```

Run `ManifestServ`, `/usr/bin/ManifestServ`, specifying the `-t` option, in order to save the temporary file, and the `-v` option, in order to provide verbose output which will have the line number of the error.

```
# ManifestServ -f /usr/share/distro_const/DC-manifest -t -v manifest_file

ManifestServ -f /usr/share/distro_const/DC-manifest -t -v my_distro.xml
/usr/share/distro_const/DC-manifest.defval.xml validates
Checking defaults for name
Checking defaults for distro_constr_params/distro_constr_flags/stop_on_error
Checking defaults for distro_constr_params/pkg_repo_default_authority/main/url
...
...
(omitted content)
...
```

EXAMPLE 2–3 Debugging Schema Validation Errors *(Continued)*

```
...
/tmp/my_distro_temp_7870.xml:350: element pair: Relax-NG validity error
: Element pair failed to validate attributes
/tmp/all_lang_slim_cd_x86_temp_7870.xml fails to validate
validate_vs_schema: Validator terminated with status 3
validate_vs_schema: Validator terminated abnormally
Error validating manifest against schema /usr/share/distro_const/DC-manifest.rng
Error running Manifest Server
schema_validate: Schema validation failed for DC manifest /tmp/my_distro_temp_7870.xml
```

The temporary file will be named near the end of the output. In the example above, the file is `/tmp/my_distro_temp_7870.xml`. Per the bold error messages, open that file and go to line 350 to find the issue. In this example, the line 350 looks like this:

```
<key_value_pairs>
    <pair value='/usr/share/distro_const/slim_cd/slimcd_iso.sort' key='iso_sort'/>
    <pair VaLuE='myvalue' key='mykey'/>
</key_value_pairs>
```

The attribute, `VaLuE`, is incorrect. This contrived example should have `value` in all lowercase letters, as shown the line immediately above that one. The second to last message line states that the schema validation fails. The schema used for validation for the distribution constructor is `/usr/share/distro_const/DC-manifest.xml`. The schema shows that the only attributes for `<pair>` are `<value>` and `<key>`, not `<VaLuE>`.

Debugging Semantic Validation Errors

Semantic validation is also done. Semantic validation checks content for "meaning" and context errors as opposed to checking only the syntax. For example, finalizer scripts listed in a manifest can be validated to confirm that the scripts are executable files.

The following shows a case where the manifest failed semantic validation.

```
# distro_const build -l my_distro_sem.xml
/usr/share/distro_const/DC-manifest.defval.xml validates
/usr/share/distro_const/grub_setup.py either doesn't exist
or is not an executable file
validate_node: Content "/usr/share/distro_const/grub_setup.py"
at img_params/output_image/finalizer/script/name did not validate
Error validating manifest tree content
```

Semantic validation employs functions to do the validation, and those functions print error messages explaining why the manifest failed validation. In this contrived case, the file `/usr/share/distro_const/grub_setup.py` is missing, and the error message points directly to the problem. In this case, either `grub_setup.py` needs to be restored, or, if appropriate, the reference to that file needs to be removed from the manifest.

You can still run `ManifestServ` with `-v` to get more details on semantic validation, but this command option will merely list the one failure among many successes, and may produce output which is harder to read than when `-v` is not specified.

Checking Data

Once validation and other preprocessing is completed, `ManifestServ` prompts for data to dump and check. This step is more useful for testing the data serving process rather than testing the data itself, since the data is plainly visible in the manifest itself.

Additional Installation Information

See the following additional resources.

TABLE 2–5 Installation Documentation

Document	Topic
Oracle Solaris 11 Express Release Notes	Additional troubleshooting information
Getting Started With Oracle Solaris 11 Express	Installing from the live CD or using the text installer
Oracle Solaris 11 Express Automated Installer Guide	Performing automated installations to multiple systems
Managing Boot Environments With Oracle Solaris 11 Express	Creating and managing multiple boot environments on your system

◆ ◆ ◆ CHAPTER 3

x86: Design and Build a Virtual Machine

You can use the distribution constructor to design and build a preinstalled Oracle Solaris Virtual Machine. See the following information.

What is a Virtual Machine?

A *Virtual Machine* is a tightly-isolated software container that can run its own operating system and applications as if it were a physical computer. A Virtual Machine (VM) behaves exactly like a physical computer. A VM contains it own virtual hardware, such as a software-based CPU, RAM, hard disk, and network interface card (NIC). The VM that you create with the distribution constructor will include an Oracle Solaris operating system preinstalled in the Virtual Machine.

VirtualBox is an x86 virtualization application which facilitates the creation of Virtual Machines.

What is the Final Product?

When the distribution constructor builds a Virtual Machine, the Virtual Machine is exported in three formats by default, and stores as a set of files in the `<build_area>/media` directory on the host system.

Note – A *host* system is the system on which you run the distribution constructor and build a Virtual Machine.

A *client* system is the system on which you deploy a Virtual Machine.

The files are stored in the following sub-directories and formats.

- `build_area/media/esx` – Virtual machine files in a format suitable for VMware's ESX/ESXi product.
- `build_area/media/ovf` – Virtual machine files in an OVF format for hypervisors that can import such a format.

Note – OVF is an industry-standard, platform-independent format for Virtual Machines.

The files in each folder are Virtual Machine images. You can burn the files to a DVD.

Note – Another end product is an XML file that records, just for reference, the settings that the distribution constructor used to configure the Virtual Machine.

System Requirements for Building Virtual Machines

In order to build a virtual machine, you must have the following set up on your host system.

TABLE 3–1 System Requirements Table

Requirement	Description
40 GB Disk space	To use the default distribution constructor and the default manifests, you need about 40 GB of free disk space in the ZFS dataset on the host system where the Virtual Machine image will be built. The actual space needed depends in part on how many packages that you choose to install in the image.
4 GB Memory	You need about 4 GB memory on the host system in order to run a VirtualBox guest, the operating system, and the distribution constructor. That total includes about 1GB for running the Virtual Machine, and about 3 GB for running the operating system and the distribution constructor.
Oracle Solaris release	Install the x86 Oracle Solaris operating system (OS) on the host system. The Oracle Solaris release version on your system must be the same as the release version of the AI images that you use with the distribution constructor. **Note –** To run the distribution constructor on your system, you will need to assume the root role by executing the `su -` command.

TABLE 3–1 System Requirements Table *(Continued)*

Requirement	Description
Network access	The distribution constructor makes use of an automated installer (AI) image and an AI client manifest in order to perform an installation inside the Virtual Machine. The host system running the distribution constructor needs to have network access to an IPS repository, such as `http://pkg.opensolaris.org`, that is specified in the AI client manifest. You can use the default AI client manifest, or you can provide a custom AI client manifest as described in "Optional: Customize Build Specifications" on page 29. By creating a custom AI client manifest, you can change installation specifications such as defining a particular installation target or modifying the list of packages to be included in the operating system.
VirtualBox, version 3.0.12	You must have VirtualBox installed on the host system.
`distribution-constructor` package	Install the most recent copy of the `distribution-constructor` package, which contains the distribution constructor application, on the host system. **Note –** You can use the Package Manager tool to install the required package. The Package Manager is available on the menu bar on the desktop of the Oracle Solaris operating system. On the menu bar, go to System>Administration>Package Manager. Alternately, use IPS commands such as the following to install this package: `# pkg install` *distribution-constructor*

How to Build a Virtual Machine

The process used to build a Virtual Machine is straightforward:

1. "Get an AI Boot Image" on page 27
2. "Optional: Customize Build Specifications" on page 29
3. "Build the Virtual Machine" on page 33

Get an AI Boot Image

A complete Virtual Machine will be created using the settings specified in the Virtual Machine manifest, `vmc_image.xml`. The Virtual Machine manifest is an XML file located at `/usr/share/distro_const/vmc/vmc_image.xml`. The `vmc_image.xml` file points to a default set of finalizer scripts that are used to implement the final media-specific settings for the Virtual Machine. In addition, an automated installer (AI) boot image is used to create the Virtual Machine.

You must provide the AI boot image. Then, you must modify the Virtual Machine manifest to point to this image.

Download an AI image and reference that image in the Virtual Machine manifest as follows.

Note – You can create a custom AI image by using the distribution constructor. However, it's not necessary to create a custom image.

▼ Providing a Boot Image

1 **Download an x86 AI image from `http://www.oracle.com/technetwork/server-storage/solaris11/downloads/index.html`.**

Note – The AI image is only about 200–250 MB.

Make sure that the release version for the AI image is the release version for the Oracle Solaris operating system that you plan to install in your Virtual Machine.

2 **Store the AI image on your local file system.**

The AI image is an .iso file, a collection of software provided in one file. When you download the image, you store it as an .iso file on your system.

Note the name and location of this file. You will have to provide that information in the manifest.

3 **Modify `vmc_image.xml` to point to this image as follows.**

Find the following XML field in the manifest:

```
<script name="/usr/share/distro_const/vmc/prepare_ai_image">
    <checkpoint
        name="prepare-image"
        message="prepare bootable ai image"/>
    <argslist>
    <!-- Path to bootable AI image ISO -->
        "/export/home/name_of_bootable_ai_image"
    <!-- What AI client manifest to use for installation.
         "default" will use the existing AI client manifest
         included on the AI media.  To use a custom
         AI client manifest, provide a path to your custom manifest -->
    "default"
 </argslist>
</script>
```

This field above references the `prepare_ai_image` finalizer script. This script finds and prepares the specified boot image for installation into the Virtual Machine and provides an option to specify a custom AI client manifest.

The following line from this script provides a default path to an AI image, using a placeholder, `name_of_bootable_ai_image`, in place of an actual filename.

```
"/export/home/name_of_bootable_ai_image"
```

Change this line to point to the location on your local system, using your filename, for the AI image that you downloaded in the prior step. For example,

```
"/export/home/myimage.iso"
```

Note – The file location is provided within quotes. Do not remove the quotes. The quotes are used to delineate each argument in the field.

You have the option to further modify this `prepare_ai_image` field to point to a custom AI client manifest as described in the next section.

Optional: Customize Build Specifications

You may choose to customize some of the other specifications in the Virtual Machine manifest file. You can make these changes by editing the XML fields in `vmc_image.xml` before you build your Virtual Machine.

The following table lists the fields in `vmc_image.xml`, provides the default values for each field, and describes which fields can be edited.

Note – Some of the fields in this table are also used in the other distribution constructor manifests. Many of these fields, however, are unique to the Virtual Machine manifest.

TABLE 3–2 Virtual Machine Manifest Fields

Manifest Field	Description
`<build_area>rpool/dc</build_area>`	The build area field specifies the area where the Virtual Machine will be created. You can use the default area if it is suitable for your system, or you can specify a different build area on the host system. **Note –** To use checkpointing to pause and resume during the build, you must specify a ZFS dataset, or a mount point that correlates to a ZFS dataset, as your build area. If the ZFS dataset does not exist, it is created during the build. The ZFS pool that you specify, however, must already exist.

TABLE 3–2 Virtual Machine Manifest Fields *(Continued)*

Manifest Field	Description

`<distro_constr_flags>`
`<stop_on_error>true</stop_on_error>`
`<checkpoint_enable>`
`true`
`</checkpoint_enable>`
	Within the `<distro_constr_flags>` tag, the value of true for `<stop_on_error>` means that, if an error occurs in the build, the distribution constructor stops running.
	
	The value of true for `<checkpoint_enable>` gives you the ability to pause at any specified steps, called checkpoints, when building the Virtual Machine, and to restart the build at any specified checkpoints. For instructions, see "Building an Image in Stages by Using Checkpoint Options" on page 18.

`<finalizer>`	The `finalizer` section contains a list of references to the finalizer scripts that are run when the Virtual Machine is built. These scripts are used to customize the image and are run in the order listed in this manifest. As shown in the following script entries, each script field includes a required `checkpoint` field with a name for the checkpoint. The checkpoint marks the stage during the Virtual Machine build when this script is run. The checkpoint field also contains a checkpoint message. When the script runs, the message will display. Each script also includes a field, `argslist`, that provides any arguments that are required for that particular script to run. You can edit these arguments as described in this table. **Note –** The arguments are provided within quotes. Do not remove the quotes. The quotes are used to delineate each argument in the field.

TABLE 3–2 Virtual Machine Manifest Fields *(Continued)*

Manifest Field	Description

`<script name="/usr/share/distro_const/vmc/prepare_ai_image">`
`<checkpoint`
`name="prepare-image"`
`message="prepare bootable ai image"/>`
`<argslist>`
`<!-- Path to bootable AI image ISO -->`
`"/export/home/name_of_bootable_ai_image"`
`<!-- What AI client manifest to use for installation.`
`"default" will use the existing AI client manifest`
`included on the AI media. To use a custom`
`AI client manifest, provide a path to`
`your custom manifest -->`
`"default"`
`</argslist>`
`</script>`
	In the prior section, you modified this `prepare_ai_image` script reference to point to the AI image that you have downloaded. You can, also, in this field, provide a pointer to a custom AI client manifest. By providing a custom AI client manifest, you can change the default installation specifications such as defining a particular installation target or changing the list of packages to be installed with the operating system.
	
	Change the variable, `"default"`, to the path for your custom AI client manifest. The path will be a file path on the system where you are running the distribution constructor, as shown in this example.
	
	`"home/user/mymanifest.xml"`
	
	For instructions about creating a custom AI client manifest, see "Creating a Custom AI Manifest" in *Oracle Solaris 11 Express Automated Installer Guide* .

`<script name="/usr/share/distro_const/vmc/create_vm">`
`<checkpoint`
`name="create-vm"`
`message="create and configure virtual machine"/>`
`<argslist>`
`"16000"`
`"1536"`
`"opensolaris"`
`</argslist>`
`</script>`
	The `create_vm` script builds and configures the Virtual Machine. The `argslist` field provides 3 arguments for the script. You can edit these arguments as follows.
	■ Virtual machine disk size – The default is 16000 MB. Values between 12000 and 99.999.999 are valid.
	■ Virtual machine ram size – The default is 1536 MB. Values between 1000 and 16,384 MB are valid.
	■ Virtual machine type – The default is `opensolaris` for a 32–bit Virtual Machine. Or, use `opensolaris_64` for a 64–bit Virtual Machine.
	
	Note – The `create_vm` script and its arguments affect the Virtual Machine specifically while it is being created. See the `post_install_vm_config` script below for arguments that affect the postinstallation Virtual Machine.

`<script name="/usr/share/distro_const/vmc/install_vm">`
`<checkpoint`
`name="install-vm"`
`message="Boot and Install virtual machine"/>`
`</script>`
	The `install_vm` script boots and installs the Virtual Machine in the build area.

TABLE 3–2 Virtual Machine Manifest Fields *(Continued)*

Manifest Field	Description

`<script name="/usr/share/distro_const/vmc/post_install_vm_config">`
`<checkpoint`
`name="post-config"`
`message="Post installation virtual machine configuration"/>`
`<argslist>`
`"1024"`
`"1"`
`"on"`
`</argslist>`
`</script>`
	The `post_install_vm_config` script performs postinstallation configuration on the Virtual Machine. The `argslist` field provides 3 arguments for the script. You can edit these arguments as follows.
	■ Virtual Machine ram size – The default is 1024 MB. Values between 1000 and 16,394 are valid.
	■ Virtual Machine number of CPUs – The default is 1 CPU. Values between 1 and 32 are valid.
	■ Virtual Machine VT-x/AMD-V support – Valid options are on or off.

`<script name="/usr/share/distro_const/vmc/export_esx">`
`<checkpoint`
`name="export-esx"`
`message="Export virtual machine for VMWare ESX"/>`
`<argslist>`
`"esx"`
`</argslist>`
`</script>`
	The `export_esx` script converts the Virtual Machine into a set of ESX files that are ready for import. If you want your Virtual Machine files to be set up in ESX format, use this script. Otherwise, comment out this script.
	For example, if you want to comment out the following field:
	`<sample_script>example</sample_script>`
	Enclose it as follows:
	`<!--`
	`<sample_script>example</sample_script>`
	`-->`
	The ESX Virtual Machine will consist of a set of files in the `build_area/media/esx` folder.
	Note – If you use both the `export_esx` and `export_ovf` scripts, identical Virtual Machines for each format are created.

`<script name="/usr/share/distro_const/vmc/export_ovf">`
`<checkpoint`
`name="export-ovf"`
`message="Export virtual machine in OVF format"/>`
`<argslist>`
`"ovf"`
`</argslist>`
`</script>`
	The `export_ovf` script converts the Virtual Machine into a set of OVF files that are ready for import. If you want your Virtual Machine files to be set up in OVF format, use this script. Otherwise, comment out this script, so it will not be run.
	The OVF Virtual Machine will consist of a set of files in the `build_area/media/ovf` folder.

Build the Virtual Machine

After you have customized the Virtual Machine manifest, you are ready to build a Virtual Machine. Run the `distro_const` utility to build the Virtual Machine, using optional checkpoints to pause, test, and resume the build as needed. For instructions about using the `distro_const` utility, see "Building an Image" on page 17.

Troubleshooting During the Build

The distribution constructor uses the automated installer to perform an installation inside the Virtual Machine. The auto installer, executing in the Virtual Machine client, is not able to communicate progress to the Virtual Machine host. Therefore, it is not possible for the distribution constructor to closely track the installation progress as it occurs inside the Virtual Machine client. When the installation is complete, the auto installer turns off the Virtual Machine, and the distribution constructor resumes its work.

Note – If the auto-installer encounters an error that results in a failed installation, the Virtual Machine is not shutdown. Then the distribution constructor appears to "hang," when it's actually waiting for the Virtual Machine to shutdown.

You can monitor the progress of the automated installation as it is executing in the Virtual Machine client. Use the `rdesktop` utility found in the standard Oracle Solaris IPS repositories, or use another remote desktop (RDP)-enabled client.

To connect to the console of a running Virtual Machine installation, do the following:

1. If it's not already installed, install the `rdesktop` package.

   ```
   # pkg install remote-desktop/rdesktop
   ```

2. Connect to the running Virtual Machine which appears to be hung.

   ```
   # rdesktop -a 16 localhost:3389
   ```

This command tells `rdesktop` to connect to the local machine on port 3389, with a 16-bit color depth for better performance.

The port number used in this command is the default port number used by the first Virtual Machine started on the host. If you are running multiple invocations of the distribution constructor, then you should look in the distribution constructor's detailed log file to see what port the Virtual Machine is running under. In the log file, you'll see lines such as:

```
==== install-vm: Boot and Install virtual machine

Invoking: VBoxHeadless startvm OpenSolaris_VM
VirtualBox Headless Interface 3.0.8
```

```
Listening on port 3389
```

The last line in the log file states that the Virtual Machine in question can be contacted on port 3389. Once you run the `rdesktop` command, a GUI window opens where you can see the console for the running Virtual Machine. You can log in to the console as follows:

```
username: jack
password: jack
```

Note – The default username and password are "jack" and "jack." The default root password is "solaris."

Once you are logged in, you can debug the installation, just like any other automated installer client installation. See Appendix A, "Troubleshooting Automated Installations," in *Oracle Solaris 11 Express Automated Installer Guide* .

Further Information

See "Additional Installation Information" on page 23.

◆ ◆ ◆ APPENDIX A

Custom Finalizer Scripts, Examples

The following are custom finalizer scripts that enable you to perform particular tasks.

Note – You will have to modify the manifest to point to any new custom scripts. See instructions in "Using Custom Scripts" on page 37.

Sample Script for Adding Packages to an Image

When putting together an image, you can experiment with how an image works when packages are added or removed from a working set. This type of experimentation is supported by the distribution constructor. Additional packages can be added to the existing list in the `package` section of a manifest file. And, packages that you want to remove can be added to the `post_install_remove_package` section. After you have made your modifications to the manifest file to add or remove packages, you need to restart the build process from the beginning, and download all the packages again. That can take time. You can make these modifications more rapidly by using finalizer scripts.

The following custom script adds an IPS package to the image from an alternate repository specified in the manifest file. The package is added to the package image area. The name of the package to add is included in the script. This example also demonstrates how to use the `ManifestRead` program to obtain values from the manifest file.

EXAMPLE A–1 Sample Script for Adding Packages

```
#!/bin/ksh
#
#
# Name:
```

EXAMPLE A–1 Sample Script for Adding Packages *(Continued)*

```
# add_my_pkg
#
# Description:
# This script will add the package SUNWcdrw from
# the alternate repository specified in the manifest
# at pkg_repo_addl_authority.
#
# Args:
#
#   5 arguments are passed in by default from the DC.
#
#   MFEST_SOCKET: Socket needed to get manifest data via ManifestRead object
#   PKG_IMG_PATH: Package image area
#   TMP_DIR: Temporary directory
#   BR_BUILD: Area where boot archive is put together (not used in this example)
#   MEDIA_DIR: Area where the media is put (not used)
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if [ "$#" != "5" ] ; then
        print -u2 "Usage: $0: Requires 5 args:"
        print -u2 "    Reader socket, pkg_image area, tmp dir,"

        print -u2 "    boot archive build area, media area"
        exit 1
fi

MFEST_SOCKET=$1

PKG_IMG_PATH=$2
if [ ! -d $PKG_IMG_PATH ] ; then
        print -u2 "$0: Image package area $PKG_IMG_PATH is not valid"
        exit 1
fi

PKGCMD="/bin/pkg"

#Install this package
TEST_PKGS="SUNWcdrw"

#
# You would have specified the additional repository like this in the manifest
#
#
#   <pkg_repo_addl_authority>
#        <main url="http://localhost:10000" authname="localtest"/>
#   </pkg_repo_addl_authority>

# Get the alternate repository URL from the manifest
add_url=/usr/bin/ManifestRead ${MFEST_SOCKET} \
"img_params/pkg_repo_addl_authority/main/url"

# Get the alternate repository authority from the manifest
add_auth=/usr/bin/ManifestRead ${MFEST_SOCKET} \
"img_params/pkg_repo_addl_authority/main/authname"
```

EXAMPLE A–1 Sample Script for Adding Packages *(Continued)*

```
added_authority=0

#
# Check if authority is already set in the package image area
# if not, add it in
#
${PKGCMD} -R $PKG_IMG_PATH authority $add_auth > /dev/null 2>& 1
if [ $? != 0 ] ; then
        ${PKGCMD} -R $PKG_IMG_PATH  set-authority -O ${add_url} ${add_auth}
        added_authority=1
fi

if [ $? != "0" ] ; then
        print -u2 "$0: Unable to set additional authority"
        exit 1
fi

for t in ${TEST_PKGS} ; do
        pkg_name="pkg://${add_auth}/${t}"
        ${PKGCMD} -R $PKG_IMG_PATH install ${pkg_name}
        if [ $? != "0" ] ; then
                print -u2 "$0: Unable to install ${pkg_name}"

                exit 1
        fi
done

# if we have added the additional authority, unset it so it doesn't pollute what's
# originally there
if [ $added_authority == 1 ] ; then
         ${PKGCMD} -R $PKG_IMG_PATH  unset-authority ${add_auth}
fi

exit 0
```

Using Custom Scripts

Once you have prepared your custom script, you must add the script reference to the manifest. Then, you're ready to use the script when building the image.

Adding Script to Manifest

You must update the manifest to point to any custom scripts that you want to use. For example, if you have created a script, `/export/home/user1/test_my_pkg`, to test your custom package, you would reference the script as follows in the finalizer section of the manifest file. The checkpoint, *my_test*, indicates where the build will restart.

```
<finalizer>
      <script name="/export/home/user1/test_my_pkg">
           <checkpoint name="my_test" message="Running my test package"/>
```

```
      </script>
      <script name="/usr/share/distro_const/pre_boot_archive_pkg_image_mod">
           <checkpoint name="im-mod" message="Image area modifications"/>
      </script>
      ........
</finalizer>
```

The custom script above is designed to add or remove packages from the package image area, so you must reference this script as the first finalizer script in the `<finalizer>` section of the manifest file. Include in this reference a checkpoint name that indicates the point in the image-building process where you want to restart the build to test your added package.

Note – This particular sample script above assumes that an alternate repository was specified in the manifest, by using the `pkg_repo_addl_authority` field. The default manifest is provided with the `pkg_repo_addl_authority` field commented out, and, thus, unused. So, remove the comment tag from this field, and update the field values to provide a valid URL and `authname`.

For example, if the additional authority is available on the `localhost` at port 10000, with the name, `localtest`, you would modify the `pkg_repo_addl_authority` field in the manifest as follows:

```
<pkg_repo_addl_authority>
<main
url="http://localhost:10000"
authname="localtest"/>
<mirror url="" />
</pkg_repo_addl_authority>
```

Running the Script

After you finish the changes to the manifest file to reference your new script, you can start building your image from the beginning once. Subsequently, if you make any changes to your package, you do not need to restart from the beginning. You can generate an image with your modified package by starting at the checkpoint that runs your script.

Note – This option assumes that the build area is in a ZFS file system. Without ZFS, you cannot use checkpointing.

The following is the sequence of commands a user would execute to repeatedly test their modifications to packages.

1. Run the build from the beginning after modifying the manifest. Perform this step just once.

   `distro_const build` *my_updated_manifest.xml*

2. Check steps that are resumable.

   `distro_const build -l` *my_updated_manifest.xml*

3. Restart from the step of your package modification. You can restart from this step as many times as you need while you are debugging your modified contents.

   `distro_const build -r` *my_test my_updated_manifest.xml*

For further information about using the `distro_const` command with checkpointing options, as demonstrated in this example, see "Building an Image in Stages" on page 18.

Sample Script for Testing Alternate Packages

When a particular package is specified in the `package` section of a manifest file, that package is installed in the package image area by the distribution constructor. If you have an alternate version of this package, such as your own private copy of a package, you could test this version of the package by using a custom finalizer script. The finalizer script would replace the previously installed version of the package with a test version from an alternate repository.

In the following custom script, an IPS repository server is running on `http://localhost:10000`. The name of the package to replace is passed as an argument. The script first uninstalls the existing version of the package, then installs the test version from the alternate repository. This custom script also demonstrates how to have arguments passed in as finalizer script arguments. Note the comments in the script that explain how the script accomplishes these tasks.

EXAMPLE A–2 Sample Script for Testing Packages

```
#!/bin/ksh
#
#
# Name:
# test_my_pkg
#
# Description:
# This script will build an image using my test package
# from my local repository.
#
# Args:
#
#   These Arguments are passed in by default from the DC.
#
#   MFEST_SOCKET: Socket needed to get manifest data via ManifestRead object \
# (not used in this example)
#   PKG_IMG_PATH: Package image area
#   TMP_DIR: Temporary directory
#   BR_BUILD: Area where boot archive is put together (not used in this example)
#   MEDIA_DIR: Area where the media is put (not used)
#   TEST_PKG: Package to replace with one from the test repo
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if [ "$#" != "6" ] ; then
        print -u2 "Usage: $0: Requires 6 args:"
```

EXAMPLE A–2 Sample Script for Testing Packages *(Continued)*

```
        print -u2 "    Reader socket, pkg_image area, tmp dir,"
        print -u2 "    boot archive build area, media area, pkg_name"
        exit 1
fi


PKG_IMG_PATH=$2
if [ ! -d $PKG_IMG_PATH ] ; then
        print -u2 "$0: Image package area $PKG_IMG_PATH is not valid"
        exit 1
fi

PKGCMD="/bin/pkg"

#The test packages are passed in as arguments to this finalizer script
#You would have specified the argument like this in the finalizer section
#to pass in the argument
#
#
#  <finalizer>
#       <script name="/my/update_my_pkg_test">
#               <checkpoint name="update-pkg" message= \
"Replaces an existing package with my own"/>
#                    <argslist>
#                       "SUNWcdrw"
#                    </argslist>
#        </script>
#  </finalizer>
#
TEST_PKG=$6

# Assume that my test package resides in
#a repository running on port 10000 of the localhost.

# Specify  alternate repository URL
add_url="http://localhost:10000"

# Specify alternate repository authority
add_auth="MY_TEST"
# Check if authority is already set in the package image area, if not,
# add it in

added_authority=0

${PKGCMD} -R $PKG_IMG_PATH authority $add_auth > /dev/null 2>& 1
if [ $? != 0 ] ; then
        ${PKGCMD} -R $PKG_IMG_PATH  set-authority -O ${add_url} ${add_auth}
        if [ $? != "0" ] ; then
             print -u2 "$0: Unable to set additional authority"
             exit 1
        fi
        added_authority=1
fi
```

EXAMPLE A–2 Sample Script for Testing Packages *(Continued)*

```
if [ $? != "0" ] ; then
        print -u2 "$0: Unable to set additional authority"
        exit 1
fi

# Remove the package that's currently in the package image area.
${PKGCMD} -R $PKG_IMG_PATH uninstall ${TEST_PKG}
if [ $? != "0" ] ; then
        print -u2 "$0: Unable to uninstall ${TEST_PKG}"
        exit 1
fi

# Install the package from test repo
pkg_name="pkg://${add_auth}/${TEST_PKG}"

${PKGCMD} -R $PKG_IMG_PATH install ${pkg_name}
if [ $? != "0" ] ; then
        print -u2 "$0: Unable to install ${pkg_name}"
        exit 1
fi

# if we have added the additional authority, unset it so it doesn't pollute what's
# originally there
if [ $added_authority == 1 ] ; then
         ${PKGCMD} -R $PKG_IMG_PATH  unset-authority ${add_auth}
fi

exit 0
```

Sample Script for Adding an SVR4 Package to an Image

If you have a package that is in SVR4 format, you can test that package in an image before you convert it into an IPS package. You can use a finalizer script to add the content of an SVR4 package to the image. See the following custom script and note the comments in the script that explain how the script accomplishes this task.

Note – You will have to modify this script to include the path to your SVR4 package. See comments in script.

EXAMPLE A–3 Sample Script for Adding SVR4 Packages

```
#!/bin/ksh
#
#
# Name:
# add_my_svr4_pkg
#
# Description:
# This script will build an image using an SVR4 package
# located at a user specified file path.
```

EXAMPLE A–3 Sample Script for Adding SVR4 Packages *(Continued)*

```
#
# Note:
# The user must modify this script and provide a valid
# path to an SVR4 package in the PKG_PATH variable.
#
# #
# Args:
#
#   5 arguments are passed in by default from the DC.
#
#   MFEST_SOCKET: Socket needed to get manifest data via ManifestRead object (not used)
#   PKG_IMG_PATH: Package image area
#   TMP_DIR: Temporary directory
#   BR_BUILD: Area where boot archive is put together (not used in this example)
#   MEDIA_DIR: Area where the media is put (not used)
#
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if [ "$#" != "5" ] ; then
        print -u2 "Usage: $0: Requires 5 args:"

        print -u2 "    Reader socket, pkg_image area, tmp dir,"
        print -u2 "    boot archive build area, media area"
        exit 1
fi

PKG_IMG_PATH=$2
if [ ! -d $PKG_IMG_PATH ] ; then
        print -u2 "$0: Image package area $PKG_IMG_PATH is not valid"
        exit 1
fi

TMP_DIR=$3

#
# Install an SVR4 packages into the package image area
#

#create an admin file for non-interactive pkgadd's

ADMIN_FILE=${TMP_DIR}/admin.$$
cat << \ADMIN_EOF > $ADMIN_FILE
mail=
instance=unique
partial=nocheck
runlevel=nocheck
idepend=nocheck
rdepend=nocheck
space=nocheck
setuid=nocheck
conflict=nocheck
action=nocheck
networktimeout=60
networkretries=3
authentication=quit
keystore=/var/sadm/security
```

EXAMPLE A–3 Sample Script for Adding SVR4 Packages *(Continued)*

```
proxy=
basedir=default
ADMIN_EOF

#
# Path to your new packages
#
PKG_PATH=<User-specified path for SVR4 package>

#
# Test package name
#
SVR4_TEST_PKG=SUNWmy-test

/usr/sbin/pkgadd -n -a ${ADMIN_FILE} -d $PKG_PATH -R ${PKG_IMG_PATH} ${SVR4_TEST_PKG}

if [ $? != "0" ] ; then
        echo "installing package failed"

        exit 1
fi

/bin/rm ${ADMIN_FILE}

exit 0
```