# Digital UNIX

## Guide to Preparing Product Kits

March 1996

Product Version: Digital UNIX Version 4.0 or higher

This manual describes the procedures for creating, installing, and managing product kits. Note that Digital provides this book on line and in PostScript format as a service to its customers. This book is not part of the Digital UNIX documentation set and cannot be ordered from Digital.

**Digital Equipment Corporation**
**Maynard, Massachusetts**

# Contents

## 3 Writing Subset Control Programs



## 4 Creating Subsets

## Examples



## Figures



## Tables

# About This Manual

This manual describes the procedures for creating, installing, and managing product kits to be installed on Digital UNIX® systems.

## Audience

This manual is primarily for kit developers responsible for creating product kits. This manual assumes you are a moderately experienced user of the Digital UNIX system.

## Organization

This manual is organized as follows:

| | |
|---|---|
| Chapter 1 | Introduction<br>Presents an introduction to the kit-building process. |
| Chapter 2 | Creating the Kit Directory Structure<br>Describes how to create the kit directory and build the product kit. |
| Chapter 3 | Creating and Managing Subset Control Programs<br>Describes how to write Subset Control Programs (SCPs) to install and manage software subsets. |
| Chapter 4 | Creating Subsets<br>Describes how to create subsets and subset control files with the `newinv` and `kits` utilities. |
| Chapter 5 | Producing Distribution Media<br>Describes how to produce a product kit on the distribution media. |
| Chapter 6 | Installing the Kit<br>Describes how to install the product kit on the target system. |
| Appendix A | Digital UNIX Standard Directory Structure<br>Describes the standard directory hierarchy of Digital UNIX systems. |
| Glossary | Defines terms used in this manual. |

## Related Documents

The printed version of the Digital UNIX documentation set is color coded to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Digital.) This color coding is reinforced with the use of an icon on the spines of books. The following list describes this convention:

| Audience | Icon | Color Code |
|---|---|---|
| General users | G | Blue |
| System and network administrators | S | Red |
| Programmers | P | Purple |
| Device driver writers | D | Orange |
| Reference page users | R | Green |

Some books in the documentation set help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the Digital UNIX documentation set.

You may find the following documents helpful when preparing product kits:

- *Release Notes*

  The content of this manual differs in the printed and HTML formats.

  Both formats contain the release notes for the Digital UNIX Version 4.0 operating system. The release notes describe problems you might encounter when working with the Digital UNIX system and possible solutions for those problems.

  The printed format also contains information about new and changed features of the operating system, as well as plans to retire obsolete features of the operating system. Obsolete features are features that have been replaced by new technology or otherwise outdated and are no longer needed.

  The HTML bookshelves contain a separate manual, *New and Changed Features*, that describes the new and changed features and plans to retire obsolete interfaces.

  The release notes are for the person installing the product and for anyone using the product following installation.

- *Reference Pages Section 8 and 1m*

  This section describes commands for system operation and maintenance. It is for system administrators. In printed format, this section is divided into two volumes.

- *Writing Device Drivers: Tutorial*

  This manual provides information for systems engineers who write device drivers for hardware that runs the Digital UNIX operating system. Systems engineers can find information on driver concepts, device driver interfaces, kernel interfaces used by device drivers, kernel data structures, configuration of device drivers, and header files related to device drivers. This manual can be helpful if you are preparing product kits for a device driver.

## Reader's Comments

Digital welcomes any comments and suggestions you have on this and other Digital UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-881-0120 Attn: UEG Publications, ZK03-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

- Mail:

  Digital Equipment Corporation
  UEG Publications Manager
  ZK03-3/Y32
  110 Spit Brook Road
  Nashua, NH 03062-9987

  A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Digital UNIX that you are using.

• If known, the type of processor that is running the Digital UNIX software.

The Digital UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate Digital technical support office. Information provided with the software media explains how to send problem reports to Digital.

## Conventions

The following typographical conventions are used in this manual:

| | |
|---|---|
| # | A number sign represents the superuser prompt. |
| `%` **`cat`** | Boldface type in interactive examples indicates typed user input. |
| *`file`* | Italic (slanted) type indicates variable values, placeholders, and function argument names. |
| **[ \| ]**<br>**{ \| }** | In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed. |
| . . . | In syntax descriptions and function definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times. |
| `cat`(1) | A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages. |

# Introduction 1

A kit is a collection of files and directories that represent one or more layered products. A kit is a standard mechanism by which layered product modifications can be delivered and maintained on a Digital UNIX system. It can be distributed on a CD–ROM, diskette, or tape for installation on the customer's system with the `setld` utility. Finally, a kit can be presented to the Digital UNIX system at various times — single-user time, multi-user time, installation time, or when setting up a RIS environment.

Before building a kit, you should consider the following questions:

- What kind of product does this kit represent? Does it run in user space or kernel space? Is it used during the initial installation and bootstrap of the Digital UNIX system?
- When will this kit be installed during the initial installation and boot of the system or after the system is up and running?

The answers to these questions determine the type of format you choose, the type of medium you use to distribute the kit, and the installation procedures that your users run when they install the kit on their systems.

This chapter can help you answer these questions. It describes the product types supported by the kit-building process and the options for packaging and installing the kit on the customer's system. It leads you through the steps involved in building kits for the various kinds of products, and it describes the installation options supported by the Digital UNIX system.

When you understand the kind of kit your are building, you can go to individual chapters of the book for detailed steps for building your particular kit.

## 1.1 Product Types

The kitting process described in this book lets you deliver layered products for the Digital UNIX system. A layered product is any software product that is not part of the base operating system. Layered products can fall into the following categories:

- User products

  These products run in user space. Commands and utilities fall into this category, as do applications such as text editors and database systems.

Users interact directly with user products, for example, through commands or window interfaces.

- Kernel products

  These products run in kernel space. Users do not directly run kernel products, but the operating system and utilities access them to perform their work. For example, a device driver is one common type of kernel product. A user runs an application or utility, which generates system requests to perform operations such as opening a file or writing data to a disk. The system determines which device driver should service this request, then calls the appropriate driver interface.

- Foreign devices

  These are kernel products that may also be installed during the initial system installation and boot. Before a system manager can make use of the peripheral devices associated with foreign device controllers, the associated device drivers must be configured into the kernel. The problem is that prior to the installation of the operating system there is no kernel and no device drivers available to handle potential kernel and user requests of these foreign devices.

  The following are examples of hardware needed during the initial installation and boot of the operating system:

  – Hardware abstract layer (HAL)

    A HAL is a hardware interface that allows a given platform to operate within the Digital UNIX system. (Currently, HAL devices are not supported as foreign devices.)

  – Graphics controller

    A graphics controller is the hardware interface between the computer and a graphics terminal.

  – Disk controller

    A disk controller is the hardware interface between the computer and a disk device. (Currently, disk devices are not supported as foreign devices.)

  – Network controller

    A network controller (when booting from the network) is the hardware interface between the computer and a network device. (Currently, network controllers are not supported as foreign devices.)

## 1.2 Kit Formats

Prior to being copied onto the distribution media, the product files are gathered into subsets. Subsets group together related files and specify whether the group is required or optional for the installation procedure. You can choose to copy the product files onto the distribution media in one of the following formats:

- `tar` format

  In `tar` format, the product files belonging to the same subset are dumped to the distribution media as a single file. During installation, the `setld` utility uncompresses the files, then moves them onto the customer's system, preserving the files' original directory structure. The `gentapes` and `gendisk` utilities can create kits in `tar` format.

- Direct CD–ROM (DCD) format

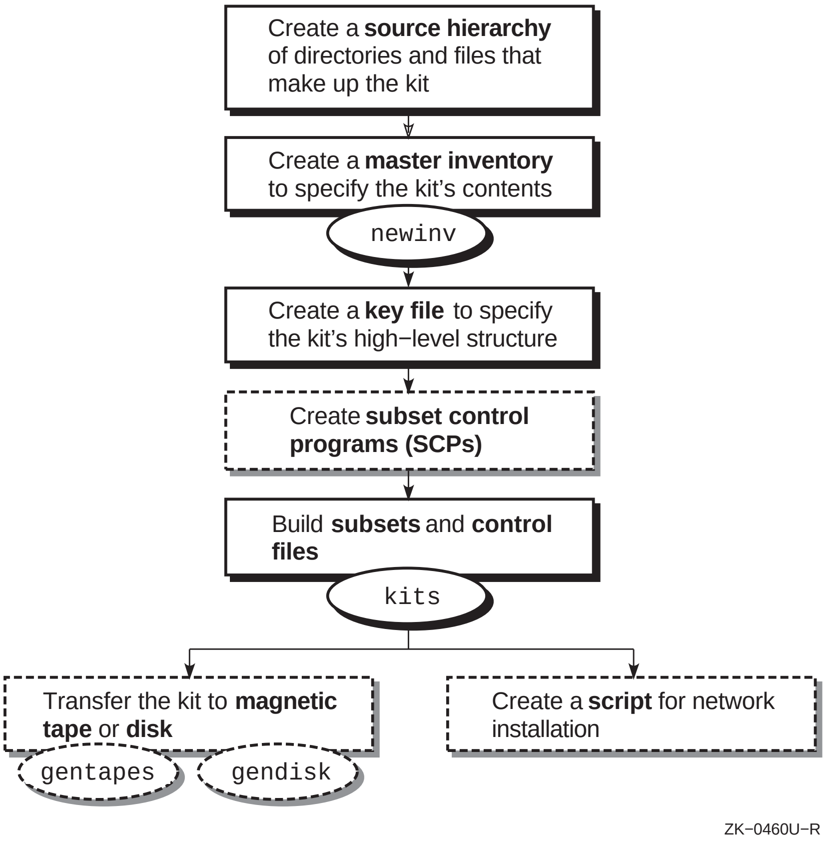
  In DCD format, the files are written to the distribution media as a UNIX file system. Subsets distributed in DCD format cannot be compressed. The `gendisk` utility can create kits in DCD format.

User and kernel products should be produced in `tar` format; foreign device products must be produced in DCD format.

## 1.3 Kit-Building Process

Figure 1-1 illustrates the process of creating and packaging a kit for use with the `setld` utility. In the figure, dashed boxes represent optional steps; for example, you do not have to create subset control programs if your kit requires no special handling when it is installed. In the figure, the commands enclosed in ellipses perform the indicated steps of the kit-building process.

**Figure 1-1: The Kit Building Process**




The kit-building process is divided into the following steps:

1. Creating the directory structure containing the source files for the kit

   On the development system, you create a hierarchy of directories and files that represent the kit you want to build. The hierarchy has the following directory structure:

   – A source hierarchy contains all the files that make up the product.

   – A data hierarchy contains files needed to build the product. A master inventory lists all the product files and the subsets in which they belong. A key file identifies the product that the kit represents.

   – An output hierarchy holds the result of the kit-building process — one or more subsets that make up the product kit.

This directory structure is the same for user products, kernel products, and foreign device products. Only the contents of these directories differs among the product types. For example, a foreign device kit needs additional files that are unique to this specific kit type.

2. Creating a subset control program

   The `setld` utility can call a subset control program to perform installation steps specific to your kit. This program is optional for user products and kernel products. You supply it on your kit only if the product requires special installation steps. The program is required for RIS installations of foreign devices.

   Most layered products require an SCP, though the actions performed by the programs differ for each product type. For example, the SCP for a kernel kit may call the `kreg` utility, while the SCP for a user product would not.

3. Building subsets and control files

   Before transferring your kit onto distribution media, you organize the product files into subsets. Subsets group together related files. For example, one subset could contain optional product files, while another subset could contain the files required to run the product.

   The `kits` utility creates subsets according to the specifications you define in master inventory and key files. The `newinv` utility can help you maintain the master inventory.

4. Producing the distribution media

   When you have created the subsets for the product, you are ready to package the kit. At this point, you must decide whether to create the kit in DCD format or in compressed or uncompressed `tar` format. This is done by selecting the appropriate packaging utilities, such as `gendisk` or `gentapes`. If you are creating a kit for a foreign device product, you must also modify the kit and add files for the `osfboot` utility's boot-link support.

5. Installing the kit

   After you have successfully created the kit, you can test the installation. For user products and kernel products, you install the kit by running the `setld` utility. For foreign device products, you use the `osfboot` utility's boot-link technology to install the kit during the initial installation and bootstrap of the system and to build a custom kernel. You may also want to install the kit on a RIS server so that other users can install it across a network.

## 1.4 Sample Products

This book uses the following sample products to demonstrate how to build kits for each product type supported by Digital UNIX:

- Orpheus Document Builder (ODB)

  This application is one of a set of applications produced by the fictitious company, Orpheus Authoring Tools, Inc. This example shows how to build a kit for a user product.

- The `/dev/none` device driver

  This peripheral device driver is introduced in *Writing Device Drivers: Tutorial*. It is produced by the fictitious company, EasyDriver, Inc. This example shows how to build a kit for a kernel product.

- The `/dev/edgd` device driver

  This graphics driver is also introduced in *Writing Device Drivers: Tutorial*. It is a device driver produced by EasyDriver, Inc. This example shows how to build a kit for a foreign device product.

# Creating the Kit Directory Structure 2

When engineers have finished developing a product, they hand over the product files to you for packaging and processing into a kit. Your first task is to organize these files by function and use, then to place them in a kit-building directory structure. When designing the kit-building directory structure, you must consider where you want to place the product files on the customer's system. You then create a kit directory structure on the development system that closely mirrors the customer's directory structure.

This chapter describes the standard directory structure of Digital UNIX systems, and it describes how to create a kit-building directory structure to fit within the standard directory structure.

**Note**

> Foreign device products require several files that are not part of the kit directory structure. These files are not installed on the customer's system, but are used by the the `osfboot` utility.

## 2.1 Designing the Customer's Directory Structure

For kit-building purposes, you can consider a product as consisting of various types of files. You can install the components of a kit in any directory on the customer's system. However, guidelines exist for deciding where to place kit files. The standard Digital UNIX system directory structure is set up for efficient organization. It separates files by function and use, as follows:

- Read-only files

  Files that are nominally read-only, such as commands, startup files (which can be modified, but not by individual users), or prototype data files are installed under `/usr/opt` in a product directory. The name of the product directory should consist of the 3-character product code and 3-digit version code. For example, the directory for the ODB product is `/usr/opt/OAT100`.

- Read/write files

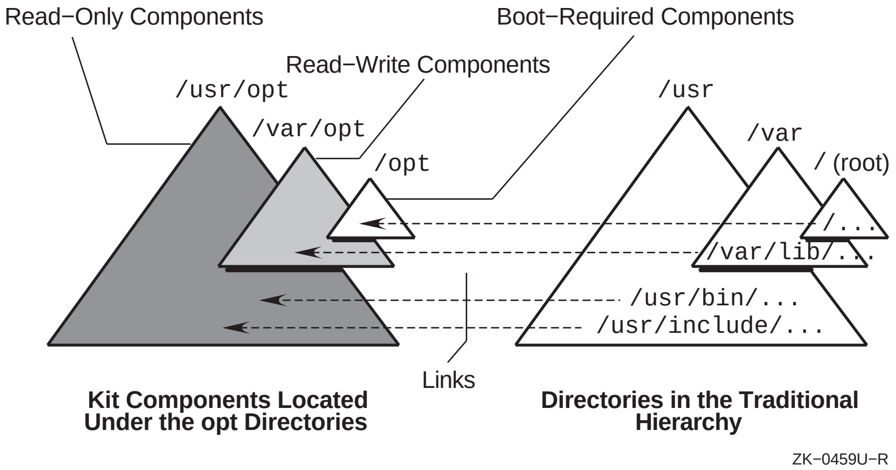
  Files that can be read and written by users, such as lists of employee telephone numbers, are installed under `/var/opt`.

- Boot files

  Files that are required at boot time, such as file system drivers, are installed under the `/opt` directory.

Figure 2-1 illustrates how a kit is installed and made available in the traditional directories.

**Figure 2-1: How Layered Products Are Installed**




Digital recommends that you design your product to be installed in the `/opt` directories. Using this standard directory structure has the following advantages:

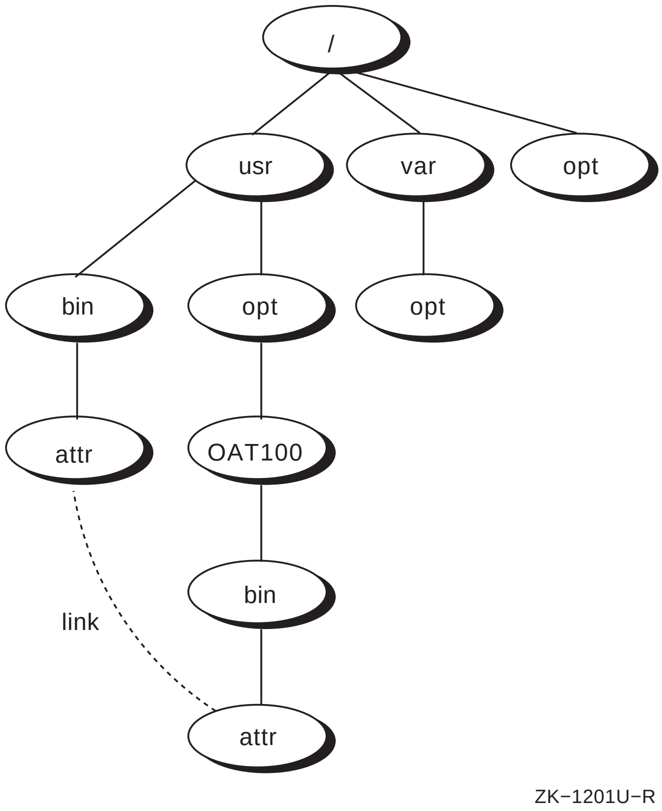
- Standardized product design and location.
- If disk partition restructuring or product maintenance becomes necessary, it is much easier to find all of your kit if its components are in the `/opt` directories rather than scattered throughout the traditional directories.
- Exporting software to share across a network is simplified and more secure; you need only export the specific directories under `/opt`, `/usr/opt`, and `/var/opt` that contain the product you want and then create links on the importing system.

  You can set up a server with multiple versions of a given product, using the links created on the client systems to determine which version a given client uses. This architecture also permits maintaining software for multiple dissimilar hardware platforms on the same server.

For users to make effective use of the file system after they install your product, files should be in directories that are in the normal search path as specified by the user's `.profile` or `.login` files, as appropriate. Your product directories would not ordinarily be in the user's search path. Therefore, the installation procedure must create links from the directory where your product files reside and the directory where users will search for them. The subset control programs create these links during product installation.

Figure 2-2 shows how the Orpheus Document Builder (ODB) product is installed in the standard directory structure, under `/opt`, `/usr/opt`, and `/var/opt`. A symbolic link makes each file accessible through the traditional directories. For example, the ODB kit's `/usr/bin/attr` command is a symbolic link to `/usr/opt/OAT100/bin/attr`.

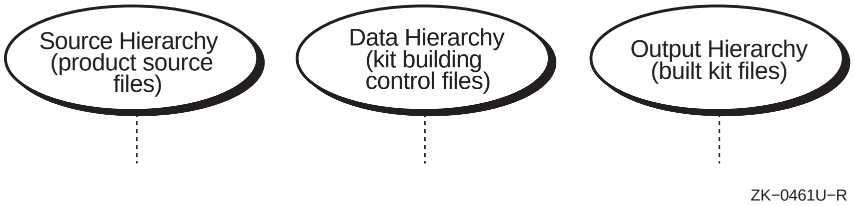
**Figure 2-2: Linking Product Files to Standard Digital UNIX Directories**



## 2.2 Kit-Building Directory Structure

To create a kit, you need three separate directory hierarchies on the kit development system, as shown in Figure 2-3.

**Figure 2-3: Kit Directory Structure**




- Source hierarchy

  The source hierarchy is a directory structure that exactly mirrors the directory structure into which your finished kit is installed by customers. You must place each file that is to become part of your kit into the appropriate directory in the source hierarchy. You can create the source hierarchy under any directory you choose.

- Data hierarchy

  The data hierarchy is a directory structure that contains the following files to specify the contents of the kit and how it is organized:

  – A master inventory file lists each of the files in the kit and defines which subset contains each file.

  – A key file specifies the kit's attributes for the `kits` command, such as the product name and version and whether the subsets are in compressed or uncompressed format.

  The data hierarchy also contains a subdirectory named `scps`; if your product includes subset control programs, these programs are stored in the `scps` directory.

  There is no specific requirement for the location of the data hierarchy, but it is good practice to place it under the same directory in which you create the source hierarchy.

- Output hierarchy

  The output hierarchy is a directory structure that contains the subsets that are placed on the kit. The subset control files that are needed during installation are stored in the `instctrl` subdirectory.

  There is no specific requirement for the location of the output hierarchy, but it is good practice to place it under the same directory in which you create the source and data hierarchies.

Create the directory structure as follows:

1. Issue the appropriate `mkdir` commands for each of the directories and subdirectories that you need.
2. Populate the `src` directory with all the files that are to be part of the finished kit. Section 2.2.1 describes the files you need for a user product. Section 2.2.2 describes the files you need for a kernel product. Section 2.2.3 describes the files you need for a foreign device product. You can choose any appropriate method for populating the source hierarchy. For example, you could create a makefile for use with the `make` command.

**Note**

File attributes, such as ownership and permissions, for all the files and directories in the source hierarchy must be exactly as they should be on the customer's system. Usually, this requirement means that you must be a superuser when populating the source hierarchy so that you can change these file attributes. Do not attempt to circumvent this requirement by setting file attributes in your subset control programs. If a superuser on the customer's system runs the `fverify` command on your subsets, attributes that the subset control programs have modified are reset to the values they have in the kit's subset inventory files.

Under most circumstances, your kit should not include any files whose pathnames exactly match those of existing system files. If you do, the kit's files are installed in place of the existing files.

### 2.2.1 Kit Directory Structure for a User Product

A user product should be designed so that the user sees it as an integral part of the system. This means that, as with base-system software, programs such as commands and utilities should be placed in directories that are part of the normal search path, such as `/usr/bin`. Similarly, libraries should be placed in directories where users would expect to find them, such as `/usr/lib`.

The example ODB kit places command files in the standard system directory (`/usr/bin`), the product's documentation in a directory created by another user product (`/usr/lib/br`), and template files for users employing the product in a directory unique to the ODB product (`/usr/share/doclib/templates`).

The actual files for the ODB kit, however, are not physically located in any of the directories listed in the preceding paragraph. The Digital UNIX system provides special locations (the `opt` directories) for optional products. Files are installed in directories under `/opt`, `/usr/opt`, and `/var/opt` so that the files are centrally located and easy to find. Then a symbolic link is created for each file that makes the file accessible through the standard system directories.

The ODB kit contains files to be installed in the following directories:

- /usr/opt/OAT100/bin
- /usr/opt/OAT100/lib/br
- /usr/opt/OAT100/share/doclib/templates

Figure 2-4 illustrates the complete directory structure for the ODB kit. In this figure, the dashed directory, dcb_tools, represents the existing directory under which you would create the source hierarchy's directories as shown.

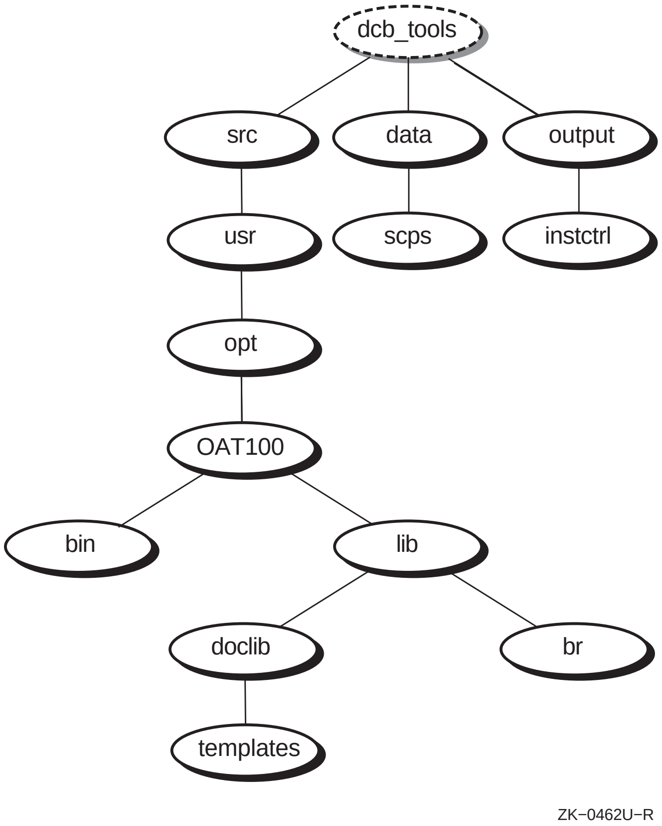
The src directory represents the root directory on the customer's system; likewise, the usr directory represents /usr on the customer's system, and all the other directories in the source hierarchy are mapped to the customer's system in the same way.

The name of the top-level product-specific directory, under the source hierarchy's opt directory, is made up of the product code and a 3-digit version number, where the first digit identifies the major version number, the second digit identifies the minor version number, and the third digit identifies the update level. For example, the product code for the ODB kit is OAT and its version number is 100, indicating major version 1, minor version 0, update 0.

File names in the standard directory structure, where users would usually expect the files to appear, are linked symbolically to the actual files installed on the customer's system. For example, the command named /usr/bin/attr exists as a link to /usr/opt/OAT100/bin/attr.

If the ODB kit included user-writable files, they would be placed under /var, and there would also be a /var/opt/OAT100... directory to contain those files. Digital recommends this convention for consistency among user products.

**Figure 2-4: Directory Hierarchy for the ODB Kit**

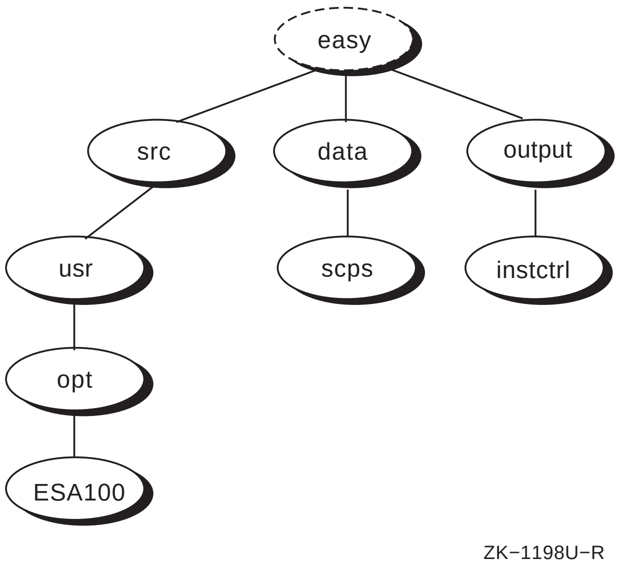



### 2.2.2 Kit Directory Structure for a Kernel Product

You set up a kit directory structure for a kernel product in the same way as you would for a user product. You create three directory hierarchies — source, data, and output — and populate the source hierarchy with the product files. Under the `src` directory, create a directory structure similar to the one on the customer's system and place the product files in `/opt`, `/usr/opt`, and `/var/opt`, as appropriate. Unlike a user product, the kit for a kernel product such as a device driver requires certain files to be present in specific directory locations.

Figure 2-5 shows the directory structure of a device driver product as it would appear in the kit development area. The driver shown here is the `/dev/none` driver produced by the fictitious company called EasyDriver, Inc. This driver is introduced in *Writing Device Drivers: Tutorial*.

**Figure 2-5: Directory Structure for the /dev/none Driver Kit**




The top-level directory (`easy`) represents the working area for all kit development at EasyDriver, Inc. The `src` directory corresponds to the customer's root directory (/). Directories under `src` have a one-to-one relationship to directories on the customer's system. The `ESA100` directory represents the top-level product directory for the `/dev/none` driver.

The files needed for building a kit depend on how the driver product will be configured on the customer's system, as follows:

- A statically configured driver is configured in the bootpath. It is built into the kernel. The kit for a statically configured driver can contain either driver source files, single binary modules, or both. A foreign device is one type of statically configured driver. However, it cannot be distributed in source form, only single binary (`.mod`) form.
- A dynamically configured driver is configured at run time. It is added into an existing kernel but is not a permanent part of the kernel. Whenever the system restarts, the system manager must reconfigure the driver. Kits for dynamically configured drivers contain only single binary modules, not the source files for the driver.

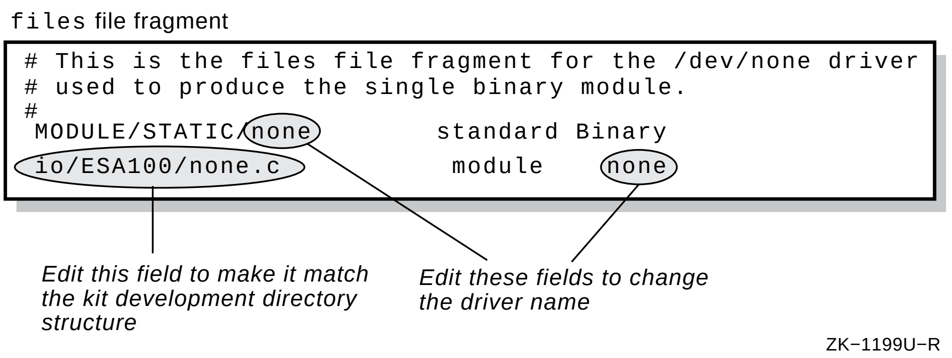
The following list describes the files that go into a device driver kit, the directories where they reside, and the types of drivers that use them:

- `files` file fragment

  Contains information about the location of the source code and modules associated with the driver, tags indicating when the driver is loaded into the kernel, and whether the source or binary form of the driver is supplied to the customer. For both statically and dynamically configured drivers, place this file in the product directory, such as `ESA100`.

  You need to edit this file if the kit development directory structure differs from the driver development directory structure or if you must change the driver name for any reason. Figure 2-6 shows which fields within the `files` file fragment need to change.

**Figure 2-6: Editing the files File Fragment**




- `sysconfigtab` file fragment

  Contains device special file information, bus option data information, and information on contiguous memory usage for statically and dynamically configured drivers. When the user installs a kernel product kit, the driver's `sysconfigtab` file fragment gets appended to the `/etc/sysconfigtab` database. You should place this file frament in the product directory, such as `ESA100`.

  You do not need to change the `sysconfigtab` file fragment unless you change the module name. The module name appears in three places within the file, as shown in Figure 2-7.

**Figure 2-7: Editing the sysconfigtab File Fragment**

Edit these attributes to point to the correct driver name.

`sysconfigtab` file fragment

```
none:
  Module_Config_Name = none
 Device_Dir = /dev
 Device_Char_Major = ANY
 Device_Char_Minor =  0
 Device_Char_Files = none
 Device_User = root
 Device_Group = 0
 Device_Mode = 666
 Device_Major_Req = Same
 TC_Option = Modname - 'None    ', Driver_Name - none,
         Type - C,  Adpt_Config - N
```

ZK−1203U−R

In this example, the driver runs on a TURBOchannel bus (indicated by the `TC_Option` entry), but a similar set of bus options would be specified for other bus types.

- *`driver`*`.mod` object module file

  Contains the single binary module for both statically and dynamically configured drivers. You should include this file in the product directory, such as `ESA100`. The subset control program references or copies this file into the customer's `BINARY` and `subsys` directories when the driver is installed.

- `*.c` (source) and `*.h` (header) files

  Contain the source code for the device driver. You should include these files in the product directory, such as `ESA100`, when the driver is statically configured and distributed in source form.

- *`driver`*`.mth` method files

  Contain driver methods that are called during autoconfiguration to create device special files for dynamically configured drivers. These files do not appear on the driver kit. The subset control program creates links to these files in the customer's `subsys` directory when the driver is installed. The device driver writer can tell you which method files the subset control program should link to, typically `/subsys/device.mth`. You need to link the method in a device driver kernel kit only if the driver needs to have device special files created for its devices.

### 2.2.3 Kit Directory Structure for a Foreign Device Product

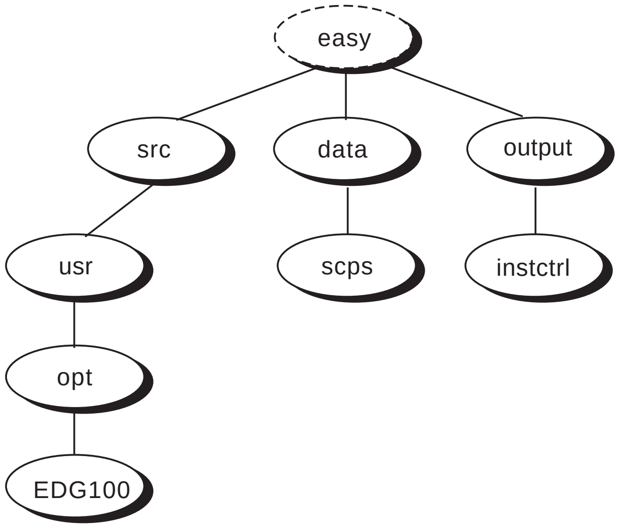
A kit for a foreign device product contains all the same files as a kit for a kernel product kit, but requires the following additional files on the media to make the media accessible during initial system installation and bootstrap:

- A *name*.kit file, where *name* is based on the device name. You provide this file on the foreign device kit to control the actions of the osfboot utility. This file configures device drivers into a temporary kernel so that system managers can use devices that would not otherwise be available during the initial installation and boot of the operating system. You place this file on the foreign device kit media after creating the kit media with the gendisk utility. Because this file is not part of the product and not installed on the system, it does not belong in the kit directory structure.
- Additional *name*.kit files for any other foreign device products on the same distribution media. A single media may contain several product kits and several *name*.kit files.
- An /etc/sysconfigtab file. This file defines attributes of the modules that osfboot configures. Like the *name*.kit file, this file does not belong in the kit directory structure.
- Links to modules in the setld portion of the kit. The osfboot utility references these linked files.
- If the foreign device kit will be installed on a RIS server, you must also supply a *kitname*.kk file in the instctrl subdirectory, where *kitname* is the product name.

Neither the *name*.kit file, the *kitname*.kk file, the /etc/sysconfigtab file, nor the links are placed on the customer's system when setld runs; the osfboot utility and the Remote Installation Services procedure use them at installation time.

Figure 2-8 shows the directory structure for the /dev/edgd graphics device driver kit.

**Figure 2-8: Directory Structure for the /dev/edgd Foreign Device Kit**




The following sections describe the contents of the *name*.kit and *kitname*.kk files.

#### 2.2.3.1 The name.kit File

Commands in the *name*.kit file describe how osfboot needs to modify the boot-link process to build this kit. When booting from a foreign device kit, osfboot sets the default directory to the kit root directory. (Ordinarily, the default is /sys/BINARY on the system disk.) Commands in the *name*.kit file indicate which modules should be added, removed, or replaced in the kernel. When you specify the modules for the foreign device driver, the directory path is relative to the kit directory, by default.

Commands in the *name*.kit file have the following form:

+[*device:*] [*/path/*]*file*.mod
: Adds *file*.mod from the root or the specified device. You can specify a full path or accept the default.

–*file*.mod
: Deletes *file*.mod from the module list on the default path for *file*.mod.

*file*.mod=[*device:*][*/path/*]*new*.mod
: Replaces *file*.mod on the default path with the module you specify.

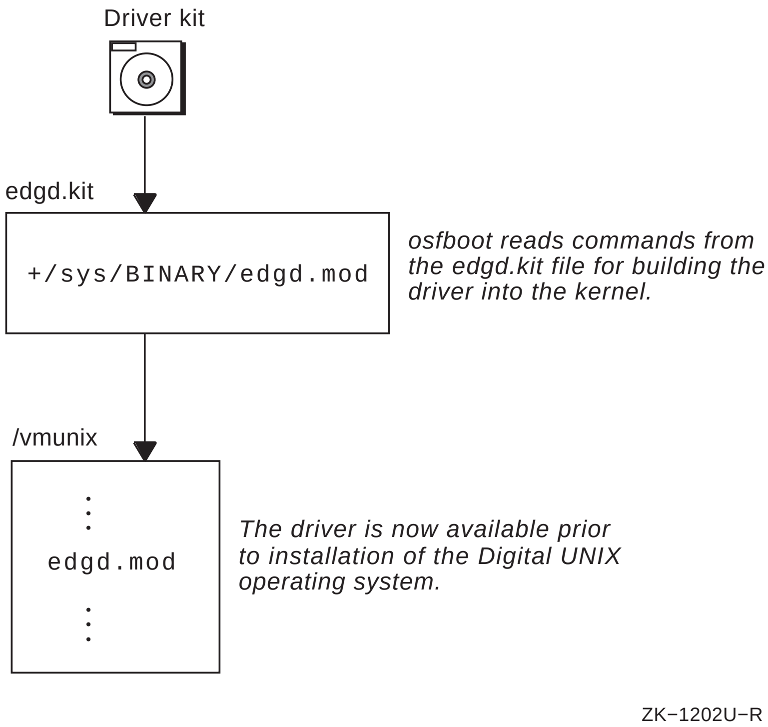
For example, the `/dev/edgd` foreign device supplies one *name*`.kit` file, which adds the `edgd.mod` single binary module to the kernel. In this example, `/sys/BINARY` refers to the directory on the kit, not the system disk:

```
+/sys/BINARY/edgd.mod
```

Figure 2-9 shows how the *name*`.kit` file works with the `osfboot` software when installing a foreign device.

In the figure, the driver kit contains a *name*`.kit` file called `edgd.kit`. The `edgd.kit` file instructs the `osfboot` utility to build and configure a temporary `/vmunix` kernel that includes the `/dev/edgd` driver. Upon completion, this temporary `/vmunix` kernel makes the `/dev/edgd` driver available to handle user and system requests of a specific hardware device prior to the installation of the Digital UNIX operating system.

**Figure 2-9: Using a name.kit File During System Installation**



#### 2.2.3.2 The kitname.kk File

The Remote Installation Services (RIS) utility provides the ability to install foreign device product kits into a RIS area for subsequent installation on a client. For more information about this feature, see *Sharing Software on a Local Area Network*.

If you want to allow the device driver product to use this feature, create a `*kitname*.kk` file in the `instctrl` directory. The file may be empty, but it must exist. It indicates to RIS that a foreign device driver kit exists.

# Writing Subset Control Programs 3

Subset control programs perform special tasks beyond the basic installation managed by `setld`. If your kit requires no special installation handling, you do not need subset control programs. The following are some of the reasons why you might need a subset control program:

- Some of your kit's files must be customized before the product will work properly.
- You want to offer the user the option of installing some of the files in a nonstandard location.
- You want to register and statically or dynamically configure a device driver.
- Your kit depends on the presence of other products.
- You need to establish nonstandard permissions or ownership for certain files.
- Your kit requires changes in system files, such as `/etc/passwd`.
- You want to provide RIS support for a foreign device product.

All of these tasks can be performed by a subset control program. Layered product kits designed according to the guidelines in Chapter 2 must have subset control programs to create the required links.

This chapter describes how to write subset control programs for layered products.

## 3.1 Common Tasks of a Subset Control Program

Regardless of the specific tasks that they perform, all subset control programs perform these common tasks:

- They are named according to certain conventions and placed in the kit building directory structure so that the `kits` utility can find them.
- They include library routines supplied by Digital UNIX.
- They are invoked at various times by the `setld` utility.
- If errors occur, they must exit and return an error status to `setld`.

- They can call routines to return subset information to global variables. By using these routines, you do not have to hard-code subset information into the subset control program.
- They can call routines to determine whether the subset control program is running in a dataless environment.

The following sections describe the common tasks that all subset control programs perform.

### 3.1.1 Creating Subset Control Program Source Files

You create one subset control program for each subset that requires special handling during installation. The program can be written in any programming language, but you must take care that your subset control program can be executed on all platforms on which the kit can be installed. If your product works on more than one hardware platform, you cannot write your subset control program in a compiled language. For this reason, Digital recommends that you write your subset control program as a script for `/sbin/sh`.

Usually subset control programs are very short programs. If written as a shell script, a subset control program should be under 100 lines in length. If your subset control program is lengthy, it is likely that you are trying to make up for a deficiency in the architecture or configuration of the product itself.

Place all subset control programs that you write in the `scps` directory, a subdirectory of the data directory. The subset control program's file name must match the subset name to which it belongs, and it must end with the `scp` suffix. For example, the ODB product defines two subsets, named `OATODB100` and `OATODBDOC100`. If both subsets required a subset control program, the source file names would be `OATODB100.scp` and `OATODBDOC100.scp`.

When you create the subsets as described in Chapter 4, the `kits` utility copies the subset control programs from the `scps` directory to the `instctrl` directory. If a subset has no subset control program, the `kits` utility creates an empty subset control program file for it in the `instctrl` directory.

### 3.1.2 Including Library Routines

Digital UNIX provides a set of routines in the form of Bourne shell script code. These routines are in the file `/usr/share/lib/shell/libscp`.

You should not copy these routines into your subset control program. Such a design would prevent your kit from receiving the benefit of enhancements or bug fixes made in future releases. Use the shell's source command to call in the routines, as follows:

```
./usr/share/lib/shell/libscp
```

### 3.1.3 Invoking Subset Control Programs

Your kit does not need to do anything to invoke its subset control program. The `setld` utility invokes it at various times, or phases. Before invoking your program, `setld` sets the `ACT` environment variable to indicate the current phase. The subset control program can perform any tasks that it needs during a phase, such as creating or deleting a file or displaying messages. Certain tasks, such as performing dependency checks or creating forward and backward links, should be performed only during specific phases, if the installation requires them.

Table 3-1 lists the phases at which `setld` calls the subset control program and the value of the `ACT` environment variable at each phase. Tasks performed during specific phases are also mentioned in the table.

**Table 3-1: Installation Phases**

| ACT Variable | Description |
|---|---|
| `M` | Before displaying the menu of subsets available for installation. At this phase, the subset control program can determine whether the subset should be included in the menu. |
| `PRE_L` | Before loading the subset. During this phase, the subset control program can perform any necessary dependency checks. |
| `POST_L` | After loading the subset. During this phase, the subset control program can create any forward or backward links that it needs and lock any subsets on which it depends. |
| `C INSTALL` | After securing the subset. During this phase, the subset control program can perform any configuration operations that are necessary. |
| `V` | When verifying the subset. During this phase, the subset control program can run an installation verification procedure. |
| `C DELETE` | Before deleting the subset. During this phase, the subset control program can perform any unconfiguration operations that may be necessary before deleting the subset. |
| `PRE_D` | Before deleting a subset. During this phase, the subset control program can reverse any actions that may have been taken during the `POST_L` phase, such as removing links and dependency locks. |

**Table 3-1: (continued)**

| ACT Variable | Description |
|---|---|
| POST_D | After deleting the subset. During this phase, the subset control program can reverse any actions that may have been taken during the PRE_L phase. |

When it enters a new phase, the setld utility sets the ACT environment variable to a value corresponding to the phase. The subset control program checks the value of the environment variable to determine what action it needs to take. In some cases, setld also passes arguments to the subset control program. The subset control program uses the argument values to further determine the actions it needs to take.

Do not include a wildcard in your subset control program's option parsing routine; write code only for the cases the subset control program actually handles. For example, the subset control programs in this chapter provide no code for several conditions under which they can be invoked. The case statements that choose an action simply exit with zero status in these undetected cases, and setld continues.

### 3.1.4 Aborting the Program

Depending on the tests it makes, your subset control program could decide at some point to abort the installation or deletion of its subset. For example, if it checks for the existence of subsets upon which your product depends and fails to find one or more of them, the subset control program can abort the process.

To abort the installation or deletion of the subset, the subset control program must return a nonzero status to setld upon exiting from the particular phase for which it was called. If the subset control program returns a status of zero (0), setld assumes that the subset control program is satisfied that the setld process should continue.

### 3.1.5 Setting Global Variables

The subset control program can use global variables to store information about the current subset. These variables are listed in Table 3-2.

**Table 3-2: List of STL_ScpInit Variables**

| Variable | Description |
|---|---|
| _SUB | Subset identifier, for example, OATODB100 |
| _DESC | Subset description, for example, Document Building Tools |
| _PCODE | Product code, for example, OAT |
| _VCODE | Version code, for example, 100 |
| _PVCODE | Concatenation of product code and version code, for example, OAT100 |
| _PROD | Product description, for example, Orpheus Authoring Tools |
| _ROOT | The root directory of the installation |
| _SMDB | The location of the subset control files, ./usr/.smdb. |
| _INV | The inventory file, for example, OATODB100.inv |
| _CTRL | The subset control file, for example, OATODB100.ctrl |
| _OPT | Set to /opt/ |
| _ORGEXT | File extension for files saved by the STL_LinkCreate routine, set to pre$_PVCODE |
| _OOPS | For dependency checking, set to NULL string |

You can call the STL_ScpInit routine to define these variables and initialize them to their values for the current subset. This routine eliminates the need to hard code subset information in your subset control program. Use STL_ScpInit in all phases except the M phase to initialize global variables. All variable names begin with an underscore (_) for easy identification.

### 3.1.6 Working in a Dataless Environment

In many environments, every computer has a copy of the operating system software on its own disk. In a dataless environment, one computer acts as a server by storing the software on its disk. Other computers, called clients, access the software across the Local Area Network (LAN) rather than from their local disks. Sharing software across the network saves disk space on each of the computers in the network.

A subset control program may need to perform differently in a dataless environment, or disallow installation of the subset on such a system. You can use the following routines to handle dataless environments:

STL_IsDataless
: Checks to see if a subset is being installed into a dataless environment.

STL_NoDataless
: Declines installation of a subset into a dataless environment.

## 3.2 Performing Tasks Associated with Installation Phases

The setld utility calls the subset control program at each of the phases listed in Table 3-1. Before calling the subset control program, setld sets the ACT environment variable to a value indicating the current phase. The subset control program uses this variable to determine what action to take. You can write the subset control program as a series of case statements, where each statement handles one phase.

Some tasks must take place during specific phases. For example, establishing dependency relationships between subsets must take place during the PRE_L phase; creating links between product files and the standard directory structure must take place during the POST_L phase.

The following sections describe the tasks that a subset control program may take in each phase.

### 3.2.1 Displaying the Subset Menu (M Phase)

At the beginning of installation, the setld utility presents a menu of subsets that it can install. Before displaying the menu, it sets the ACT environment variable to M and calls the subset control program for each subset. At this time, the subset control program can determine if it will permit its subset to be offered in the menu. The subset control program should return a value of 0 (zero) if the subset can be included in the menu.

When it calls the subset control program during this phase, the setld utility passes one argument, which can have one of two values:

- –l indicates that the operation is a subset load.
- –x is reserved by Digital for extraction of the subset into a RIS server's product area.

For example, during this phase the subset control program can verify that the subset is being installed on the correct hardware platform, by issuing the machine(1) command. If the command returns a nonzero status, the subset control program exits with a nonzero status.

When setld extracts a subset into a RIS server's product area, the server actually executes the subset's subset control program to make use of the subset control program's code for the M phase of installation. You should code the M phase to detect the difference between extraction of the subset into a RIS area and loading of the subset for use of its contents.

To make this determination, check the value of the $1 command argument (either –x for RIS extraction or –l for loading). For RIS extraction, the subset control program should do nothing during the M phase. When loading subsets, it should make this machine test. The following Bourne shell example illustrates one way to code the M phase. The subset control program checks to see if it is running on Digital's Alpha processor:

```
case $ACT in
  M)
    case $1 in
      -l)
        [ "‘/bin/machine‘" = alpha ] || exit 1
        ;;
    esac
    ;;
   .
   .
   .
esac
```

Installation for a dataless client requires that the client's local copy of the machine command be used even though the installation is being performed in the dataless area on a different platform. Because the machine command is a shell script, it can be executed on any platform.

### 3.2.2 Before Loading the Subset (PRE_L Phase)

After presenting the menu and before loading the subset, the setld utility sets the ACT environment variable to PRE_L and calls the subset control program for each subset. At this time, the subset control program can take any action required to protect existing files.

For example, the subset might contain files with names duplicating those of existing files. Duplicating existing file names is usually considered a poor practice. One example of a situation in which you might do so is for the installation of a kit that contains binary files that would usually be installed by other kits but which must be different when your kit is installed.

The subset control program should also check for subset dependencies at this time. Subset dependencies are conditions under which a subset depends on the existence of one or more other subsets. Because setld can both install and remove subsets, the system administrator could attempt to remove one or more subsets on which your product depends. Because those subsets do not in turn depend on your product's subsets, setld usually will remove them without question, leaving your product unable to work. You can prevent this inadvertent destruction of your product's environment by locking the subsets on which your subset depends. Subset locking can occur during the POST_L phase.

To make dependency management easier to implement, Digital provides a set of routines in the form of Bourne shell script code. These routines are in the file /usr/share/lib/shell/libscp.

The dependency management routines use dependency expressions to examine conditions on the system. A dependency expression is a postfix logical expression that describes the conditions on which the subset depends. Dependency expressions are recursive left to right and processed using conventional postfix techniques. Dependency expressions are defined in Backus-Naur form, as follows:

| | | |
|---|---|---|
| *depexp* | **::=** | *wc_subset_id* |
| | **\|** | *depexp* **not** |
| | **\|** | *depexp depexp* **and** |
| | **\|** | *depexp depexp* **or** |

*wc_subset_id*
Represents a subset identifier that can contain file name expansion characters (asterisks, question marks, or bracketed sets of characters) as in the following example:

```
OAT[RV]DOA*2??
```

*and* operator
Requires two dependency expressions. The dependency is satisfied if both expressions are satisfied.

*or* operator
Requires two dependency expressions. The dependency is satisfied if at least one of the expressions is satisfied.

*not* operator
Requires one dependency expression. The dependency is satisfied if the expression is not satisfied.

The following are valid dependency expressions:

```
SUBSETX??0
SUBSETY200 not
SUBSET[WX]100 SUBSETY200 and
SUBSETX100 SUBSETY200 or
SUBSETX100 SUBSETY200 and SUBSETZ300 or not
```

The last of these expressions evaluates as follows:

- The and operator is satisfied if both SUBSETX100 and SUBSETY200 are present.
- The or operator is satisfied if the and operator was satisfied or if SUBSETZ300 is present.

- The not operator is satisfied only if the combination of SUBSETX100 and SUBSETY200 is not present and SUBSETZ300 is not present.

You can call the following routines to perform dependency checking:

STL_DepInit
: Establishes objects used by the STL_DepEval routine. Before you use STL_DepEval to check your subset's dependencies, you must execute STL_DepInit once. This routine has no arguments and returns no status.

STL_DepEval *depexp*
: Evaluates the dependency expression that you specify as an argument. You can use as many invocations of STL_DepEval as you need to verify that all your subset dependencies are met.

### 3.2.3 After Loading the Subset (POST_L Phase)

After loading the subset, the setld utility sets the ACT environment variable to POST_L and calls the subset control program for each subset. At this time the subset control program can make any modifications required to subset files that are usually protected from modification when the installation is complete, such as moving them to a different location. The subset control program should create links and perform subset dependency locking at this time.

As indicated in Chapter 2, a layered product's files should be installed in the /usr/opt and /var/opt areas and accessed by means of symbolic links in the standard UNIX directory structure, such as /usr/bin. These symbolic links, referred to as forward links, must be created during the POST_L phase, after the referent files are in place. Do not try to create these links during the C INSTALL phase because the /usr file system is not guaranteed to be writeable at that time. If your product includes links in /var, create these links also in POST_L. To maintain symmetry, you must remove links during PRE_D, not during C DELETE.

Symbolic links for layered products are usually created in the standard UNIX directories to refer to files that are actually in the layered product areas, /usr/opt and /var/opt. These links are relatively straightforward.

Under certain circumstances, however, you might need to create links within your product's directories in the layered product areas that refer to files in the standard hierarchy. Such backward links must be created carefully because the layered product directories can themselves be symbolic links. This means that you cannot rely on knowing in advance the correct number of directory levels (../) to include in the ln commands for your backward links. For example, /var is frequently a link to /usr/var.

When a kit is installed on an NFS server, all the backward links are made in the server's kit area. Then, when that area is exported to clients, the links are already in place for the client. Clients do not need to create any backward links.

**Note**

> NFS clients importing products with backward links must have directory hierarchies that exactly match those on the server. Otherwise, the backward links fail.

#### 3.2.3.1 Creating Forward Links

You create forward links during the `POST_L` phase. To create the link, you must first call the `STL_ScpInit` routine to initialize global variables, then call the `STL_LinkCreate` routine. The `STL_ScpInit` routine looks in the master inventory file for any entries flagged for linking. The entry in the master inventory indicates where the file has been installed. The `STL_LinkCreate` routine creates a link to that file from a directory within the standard directory structure, such as `/usr/bin`. Thus, for the `STL_LinkCreate` routine to work correctly, you must specify the files that require symbolic links by setting the link bit in the master inventory file. (See Chapter 4 for more information on the master inventory file.)

Any nonempty directories in the inventory should leave the link bit unset (set to 0) to maximize the performance of `STL_LinkCreate`. See Example 3-2 for the creation and removal of symbolic links.

The following routines creates forward links:

`STL_LinkInit`

: Used in the `POST_L` phases to establish an internal variable used by the `STL_LinkCreate` routine. Before you use `STL_LinkBack` to create a link, you must execute `STL_LinkInit` once.

  This routine has no arguments and returns no status.

`STL_LinkCreate`

: Creates forward links from the installed system to the product areas, such as the `/opt` areas. Call `STL_ScpInit` first to initialize required global variables. A forward link from the system to the product areas (under `/usr/opt` or `/var/opt`) is created for each file whose link flag is set in the master inventory file. For example, if in the master inventory, the link bit of the `./usr/opt/OAT100/bin/attr` file is set:

```
4       ./usr/opt/OAT100/bin/attr              OATODB100
```

After `STL_LinkCreate` runs, a symbolic link from `./usr/bin/attr` points to `../../usr/opt/OAT100/bin/attr`.

If a file already exists in the same name space, `setld` saves it before the link takes its place. In the previous example, if a `./usr/bin/attr` file already exists, it is saved to `./usr/bin/attr.preOAT100` before the link gets created.

All links are created relative to the install root and are dataless safe.

#### 3.2.3.2 Creating Backward Links

The subset control program should create backward links, not the link management program, so that installation on an NFS client cannot overwrite any existing backward links in the server's kit areas. (You do not run the subset control program on an NFS client.) Your subset control program should create and remove backward links in the `POST_L` and `PRE_D` phases, respectively.

Use the `STL_LinkInit` and `STL_LinkBack` routines to create backward links as follows, and use the `rm` shell commmand to remove them:

`STL_LinkInit`

Used in the `POST_L` phase to establish an internal variable that the `STL_LinkBack` routine uses. Before you use `STL_LinkBack` to create a link, you must execute `STL_LinkInit` once.

This routine has no arguments and returns no status.

`STL_LinkBack` *`link_file file_path link_path`*

Creates a valid symbolic link from your product area (under `/usr/opt` or `/var/opt`) to a directory within the standard UNIX directory structure. You can use `STL_LinkBack` repeatedly to create as many links as required. The routine takes the following arguments:

1. *`link_file`* — The name of the file to link
2. *`file_path`* — The dot-relative path of the directory where the file actually resides
3. *`link_path`* — The dot-relative path of the directory where the link should be placed

This routine returns no status.

Example 3-1 uses `STL_LinkInit` and `STL_LinkBack` in the `POST_L` phase to create a link named `/usr/opt/OAT100/lib/odb_users` that refers to the real file `/etc/odb_users`, and removes the link in the

PRE_D phase.

### Example 3-1: Example of Backward Link Creation

```
#! /sbin/sh

case $ACT in
   .
   .
   .
  POST_L)
    STL_LinkInit
    STL_LinkBack odb_users ./etc ./usr/opt/OAT100/lib
    ;;

  PRE_D)
    rm -f ./usr/opt/OAT100/lib/odb_users
    ;;
esac
```

#### 3.2.3.3 Locking Subsets

Every subset in the system's inventory has two lock files:

- A lock file indicating successful installation of a subset, named *subset-id*.lk
- A lock file indicating failed corrupt installation of a subset, named *subset-id*.dw

When a subset is installed, setld creates one of these two lock files. At that time, the lock file is empty. Assuming successful installation, that subset is then available for dependency checks and locking performed on behalf of subsets installed later. A subset's lock file can contain any number of records, each naming a single dependent subset.

For example, the ODB kit requires that some version of the Orpheus Authoring Tools base product be installed in order for the ODB product to work properly. Suppose that the OATBASE200 subset is present. When setld installs the OATODB100 subset from the ODB kit, it inserts a record containing the subset identifier OATODB100 into the OATBASE200.lk file. When the system administrator uses setld to remove the OATBASE200 subset, setld checks OATBASE200.lk and finds a record indicating that OATODB100 depends on OATBASE200. Then setld displays a warning message with this information and requires confirmation that the user really intends to remove the OATBASE200 subset.

If the administrator removes the OATODB100 subset, setld removes the corresponding record from the OATBASE200.lk file. Thereafter, OATBASE200 can be removed without flagging a dependency warning.

You can call the following routines to lock subsets:

`STL_LockInit`

Used in the `POST_L` and `PRE_D` phases to establish objects used by the `STL_DepLock` and `STL_DepUnLock` routines. Before you use `STL_DepLock` or `STL_DepUnLock` to manipulate subset locks, you must execute `STL_LockInit` once.

Because locking and unlocking are managed by different invocations of your subset control program, `STL_LockInit` must appear in both the `POST_L` and `PRE_D` phases. You should code two instances of `STL_LockInit` rather than calling it once before you make a decision based on the value of the `ACT` environment variable.

This routine has no arguments and returns no status.

`STL_DepLock` *`depexp ...`*

Used in the `POST_L` phase to add the new subset's name to the lock lists for each of the subsets named as arguments. (You can use dependency expressions as arguments.) The name of the new subset is the first argument to `STL_DepLock`. For example, the following line places `OATODB100` in the `OATTOOLS100.lk` and `OATBASE2??.lk` files:

```
STL_DepLock OATODB100 OATTOOLS100 OATBASE2??
```

### 3.2.4 After Securing the Subset (C INSTALL Phase)

After securing the subset, the `setld` utility sets the `ACT` environment variable to `C` and calls the subset control program for each subset, passing `INSTALL` as an argument. At this time, the subset control program can perform any configuration operations required for product-specific tailoring. For example, a kernel kit would statically or dynamically configure a device driver at this point. The subset control program cannot create a layered product's symbolic links at this time.

The `setld` utility enters this phase at the following times:

- When the user invokes it with the `–c` option.
- When the user invokes it with the `–l` option and without the `–D` flag to specify an alternate root directory.

The utility does not pass through this phase if the user loads the subset and specifies an alternate root directory with the `–D` flag.

### 3.2.5 Subset Verification (V Phase)

When the user invokes `setld` with the `–v` option, the utility sets the `ACT` environment variable to `V` and calls the subset control program for each subset. At this time the subset control program can perform tests to verify that the subset is installed correctly.

The `setld` utility verifies the size and checksum information for each file in the subset during loading (when the user invokes `setld` with the `–l` option). Therefore, the `setld` utility does not call the subset control program for verification during the installation process. However, in a kit containing multiple subsets, the last subset control program to be called could execute an installation verification program (IVP) or suite of IVPs to ensure that the product works properly.

### 3.2.6 Before Deleting a Subset (C DELETE Phase)

When the user invokes the `setld` utility with the `–d` option, the utility sets the `ACT` environment variable to `C` and calls the subset control program for each subset, passing `DELETE` as an argument. At this time, the subset control program can make configuration modifications to remove evidence of the subset's existence from the system. For example, a kernel kit would unconfigure a statically or dynamically configured driver during this phase. The subset control program cannot remove a layered product's links at this time.

### 3.2.7 Before Deleting a Subset (PRE_D Phase)

When the user invokes `setld` with the `–d` option, the utility sets the `ACT` environment variable to `PRE_D` and calls the subset control program for each subset. At this time, the subset control program can reverse modifications made during the `POST_L` phase of installation, such as removing dependency locks and links, or restoring moved files to their default installation locations so that `setld` can delete them properly. A return status of 0 (zero) allows the delete operation to continue.

You can call the following routines to remove links and unlock subsets:

`STL_LinkRemove`

Removes links created by `STL_LinkCreate` and restores any original files saved by `STL_LinkCreate`. Call `STL_ScpInit` first to initialize required global variables.

Modified links cannot be removed by using `STL_LinkRemove`.

`STL_DepUnLock` *`depexp ...`*

Removes the new subset's name from the lock lists for each of the subsets named as arguments.

### 3.2.8 After Deleting a Subset (POST_D Phase)

After deleting a subset, the setld utility sets the ACT environment variable to POST_D and calls the subset control program for each subset. At this time the subset control program can reverse any modifications made during the PRE_L phase of installation.

## 3.3 Using Control File Flag Bits

As explained in Table 4-3, you can use bits 8 to 15 of the subset control file's FLAGS attribute to specify special subset-related information. The subset control program can read these bits from the subset control file into which this information is placed when the kit is built. During installation, the setld utility moves the subset control file to the ./usr/.smdb. directory, where the subset control program can read the file as needed.

Not all subset control programs need to use the subset control file. It can be a convenient way to pass information between subsets, if such communication is necessary. If you must use the subset control file, do so with extreme care. Bits 0 through 7 of the FLAGS attribute are reserved by the setld utility, and you should not use or modify these bits in any way.

To find the current settings of the FLAGS attribute, the subset control program should read the subset control file, looking for a line that lists the settings. For example, the OATODBDOC100.ctrl file contains the following line:

```
FLAGS=34816
```

The value of the FLAGS attribute is expressed as a decimal integer. You can use the BitTest shell routine, contained in the file /usr/share/lib/shell/BitTest, to test an individual bit. The following example tests bit 11 of the FLAGS attribute for the OATODBDOC100 subset:

```
#! /sbin/sh

. /usr/share/lib/shell/BitTest

flags=‘sed -n ’/FLAGS=/s///p’ usr/.smdb./OATODBDOC100.ctrl‘
BitTest $flags 11 && {
   .
   .
   .
}
```

## 3.4 User Product Subset Control Program

Example 3-2 shows a subset control program for the ODB product. This program illustrates one correct method for obtaining the value of the ACT environment variable. It uses the value of the variable to determine what actions to perform, as follows:

- Performs dependency checking in the PRE_L phase to make sure the base tools are already installed
- Creates symbolic links and locks subsets on which it depends during the POST_L phase
- Notifies the user that installation is complete during the C INSTALL phase
- Removes symbolic links during the PRE_D phase
- Unlocks subsets during the POST_D phase

The program does not handle the V phase or the C DELETE phase. When setld invokes the program at these times, the program simply exits with a success status.

### Example 3-2: Subset Control Program for the ODB Product

```
#!/sbin/sh
#
# Subset Control Program for OATODB??? subset

# INCLUDE SCP LIBRARY FUNCTIONS

[ ‘/bin/machine‘ = alpha ] &&
    ./usr/share/lib/shell/libscp 1

# BEGIN EXECUTION HERE

case $ACT in 2

M) 3
    case $1 in
    -l)
        # hardware platform check
        ["‘./bin/machine‘" = alpha ] || exit 1
        ;;
    esac
    ;;

PRE_L) 4
    # dependency checking
    STL_ScpInit
    STL_DepInit

    STL_DepEval ${_PCODE}TOOLS??? ||
    {
        _OOPS="$_OOPS
```

## Example 3-2: (continued)

```
    Orpheus Authoring Tools (${_PCODE}TOOLS)"
    }

    STL_DepEval ${_PCODE}BASE[2-9]?? ||
    {
        _OOPS="$_OOPS
    Orpheus Authoring Base Tools, Version 2.0 or later (${_PCODE}TOOLS)"
    }

    [ "$_OOPS ] &&
    {
        echo "
The $_DESC require the existence of
the following uninstalled subset(s):
$_OOPS

Please install these subsets before retrying the installation.
" >&2
        exit 1
    }
    ;;
POST_L) 5
    # create symbolic links
    STL_ScpInit
    STL_LinkCreate

    # dependency locking
    STL_LockInit
    STL_DepLock $_SUB ${_PCODE}TOOLS??? ${_PCODE}BASE[2-9]?? and
    ;;
C) 6
    STL_ScpInit
    case $1 in
    INSTALL)
        echo "
Installation of the $_DESC ($_SUB)
subset is complete.

Before using the tools in this subset, please read the README.odb
file located in the /usr/lib/br directory, for information on the
kit’s contents and for release information.
"
        ;;
    esac
    ;;
PRE_D) 7
    # remove symbolic links
    STL_ScpInit
    STL_LinkRemove
    ;;
POST_D)    8
    # dependency unlocking
    STL_ScpInit
    STL_LockInit
    STL_DepUnLock $_SUB ${_PCODE}TOOLS??? ${_PCODE}BASE[2-9]?? and
    ;;

esac
```

**Example 3-2: (continued)**

```
exit 0  9
```

1. Reads in the subset control program library routines if the installation is running on an Alpha platform.
2. Examines the ACT environment variable to select the action the subset control program takes when called by setld.
3. For the M phase, if the installation is running on an Alpha platform, allows setld to continue. If not, the subset control program returns a nonzero status and exits. As a result, setld does not present this subset in its menu of subsets to be installed.
4. During the PRE_L phase, ensures that subsets on which the the OATODB100 subset depends are installed. If they are not installed, the subset control program describes the missing subsets and returns a nonzero status to setld, which aborts the installation of this subset. If multiple subsets are being installed, each is treated individually. The $_PCODE, $_OOPS, and $_DESC variables are defined by the STL_ScpInit routine.
5. During the POST_L phase, creates symbolic links from the subset by invoking the STL_ScpInit and STL_LinkCreate routines.

   After creating the links, the subset control program secures the subset by locking the subsets on which it depends to ensure that they are not deleted without warning the user of potential problems. The subset control program uses the $_SUB and $_PCODE global variables to define the subsets in the dependency relationship.
6. During the C phase, checks to see if the argument passed in by setld has the value of INSTALL. If so, the program displays a message indicating that the installation is complete. It uses STL_ScpInit and global variables to substitute the product description ($_DESC) and subset ID ($_SUB) within the message text.
7. During the PRE_D phase, removes the symbolic links created by STL_LinkCreate during the POST_L phase.
8. During the POST_D phase, unlocks the subsets on which OATODB100 depends. The $_SUB variable is defined by the STL_ScpInit routine.
9. Ensures that the subset control program returns a success status to setld for each successful action and for all of the possible cases that the subset control program does not handle. Do not code exit 0 statements elsewhere in your subset control program.

## 3.5 Kernel Product Subset Control Program

When building subset control programs for a kernel product, such as a device driver, you can choose one of the following strategies:

- Write one subset control program for a kit that contains the software subset associated with the single binary module for a statically configured driver.
- Write one subset control program for a kit that contains the software subset associated with the single binary module for a dynamically configured driver.
- Write one subset control program for a kit that contains the software subsets associated with the device driver that can be statically or dynamically configured.

Example 3-3 shows the subset control program for the single binary module associated with the `/dev/none` driver. This single binary module can be statically or dynamically configured into the kernel.

**Example 3-3: The Subset Control Program for the /dev/none Driver**

```
#!/sbin/sh
#
#
#  NONE.scp - Install the files associated with the /dev/none
#  device driver.  This driver, implemented as a single binary
#  module (.mod file), can be statically or dynamically configured
#  into the kernel.
#

case "$ACT" in 1
C)
    case $1 in
    INSTALL)  2
        echo "***** /dev/none Product Installation Menu *****"
        echo "*****                                     *****"
        echo "1. Install the static device driver subset."
        echo "2. Install the dynamic device driver subset."

        echo" Type the number for your choice [] "

        read answer
        case ${answer} in
            1)  3
            # Register the files associated with the static
            # /dev/none device driver product.
            kreg -l EasyDriverInc ESANONESTATIC100 /usr/opt/ESA100 4

            # Add the files associated with the statically configured
            # /dev/none device driver product to the customer's
            # sysconfigtab database
            sysconfigdb -a -f /usr/opt/ESA100/sysconfigtab none 5
```

**Example 3-3: (continued)**

```
            # Reminder 6
            echo "The /dev/none device driver is installed on your
            echo "system.  Before your utilities can make use of
            echo "the driver, you must build a new kernel by running
            echo "doconfig."
            ;;

            2)  7
            # Add the files associated with the dynamically configured
            # /dev/none device driver product to the customer's
            # sysconfigtab database
            sysconfigdb -a -f /usr/opt/ESA100/sysconfigtab none 8

            # Copy the none.mod file to the /subsys directory. Create
            # the none.mth driver method by linking to device.mth
            # /subsys/none.mth -> /subsys/device.mth
            cp /usr/opt/ESA100/none.mod /subsys/none.mod 9
            ln -s /subsys/device.mth /subsys/none.mth 10

            # Load the /dev/none device driver and create the device
            # special files
            sysconfig -c none 11

            echo "The /dev/none device driver was added to your
            echo " global sysconfigtab database." 12
            ;;
        esac
        ;;

    DELETE) 13
        echo "***** /dev/none Product Installation Menu *****"
            echo "*****                                 *****"
        echo "1. Delete the static /dev/none device driver subset."
        echo "2. Delete the dynamic /dev/none device driver subset."

        echo" Type the number for your choice [] "

        read answer
        case ${answer} in
            1)
            kreg -d ESANONESTATIC100 14

            # Delete the /dev/none device driver's entry from the
            # sysconfigtab database

            sysconfigdb -d none 15
            echo "The /dev/none device driver is no longer on the
            echo "system.  Remember to build a new kernel to remove
            echo "the /dev/none driver functionality." 16
            ;;

            2)
            # Make sure the /dev/none device driver is not currently
            # loaded
            sysconfig -u none  17

            # Delete the /dev/none device driver's entry from the
            # sysconfigtab database
            sysconfigdb -d none 18
```

**Example 3-3: (continued)**

```
            ;;
        esac
        ;;
    esac
    ;;
esac
exit 0
```

1 Examines the ACT environment variable to select the action the subset control program should take.

2 During the C INSTALL phase, displays a menu of installation options. The user can choose to install the driver for static configuration or dynamic configuration.

3 If the user chooses menu item 1, performs static configuradtion.

4 Registers the driver files with the kernel, using the kreg utility. The kreg utility registers a device driver product by creating the /usr/sys/conf/.product.list file on the customer's system. This file contains registration information associated with the static device driver product.

The SCP calls kreg with the following arguments:

– The –l flag

This flag indicates that the subset was loaded and it directs kreg to register the device driver product as a new kernel extension.

– Company name

The company name is EasyDriverInc. The kreg utility places this name in the company name field of the customer's /usr/sys/conf/.products.list file.

– Software subset name

The software subset name for this device driver product is ESANONESTATIC100. The subset name consists of the product code, subset mnemonic, and 3-digit version code. The kreg utility extracts information from the specified subset data and loads it into the customer's /usr/sys/conf/.products.list file.

– Directory name

The directory on the customer's system where kreg copies the files associated with this driver product is /usr/opt/ESA100. The kreg utility places this directory in the driver files path field of the customer's /usr/sys/conf/.products.list file.

5 Adds the sysconfigtab file fragment for the statically configured driver to the system's /etc/sysconfigtab database by calling the sysconfigdb utility with the following arguments:

– The `–a` flag

  This flag causes `sysconfigdb` to add the device driver entry to the customer's `/etc/sysconfigtab` database.

– The `–f` flag

  This flag precedes the name of the `sysconfigtab` file fragment whose device driver entry is to be added to the `/etc/sysconfigtab` database. This flag is used with the `–a` flag.

– The `sysconfigtab` file fragment

  The kit developer at EasyDriver, Inc. specifies the path `/usr/opt/ESA100/sysconfigtab` to indicate the location of the `sysconfigtab` file fragment for the `/dev/none` device driver.

– Device driver name

  The kit developer at EasyDriver, Inc. specifies `none` as the name of the driver whose associated information is added to the `/etc/sysconfigtab` database. This name is obtained from the `Entry_Name` attribute of the `sysconfigtab` file fragment, as described in *Writing Device Drivers: Tutorial*.

**6** Displays a message on the console terminal after the files contained on the kit are copied to the appropriate directory.

**7** When the user chooses menu item 2, performs dynamic configuration.

**8** Calls `sysconfigdb` to add the driver's `sysconfigtab` file fragment to the system's `/etc/sysconfigtab` database.

**9** Copies the dynamically configured driver's single binary module (`.mod` file) to the `/subsys` directory.

**10** Creates a symbolic link from the `/subsys/device.mth` file to the driver's `/subsys/none.mth` file.

**11** Reconfigures the system to include the `/dev/none` driver, by calling the `sysconfig` utility with the `–c` option. The `–c` option causes the `sysconfig` utility to dynamically configure the driver into the running system and create device special files. The name of the driver as specified in the `sysconfigtab` file fragment follows the option.

**12** Displays a message notifying the user that the driver has been added to the system.

**13** During the `C DELETE` phase, the subset control program displays a menu of options for deleting subsets. The user must tell the `setld` utility whether the subset to be deleted represents a statically configured driver or a dynamically configured driver. The way the driver was configured determines how the driver is deleted.

**14** If the user chooses menu option 1 (delete a statically configured driver), calls the `kreg` utility to deregister the driver with the kernel. When the `kreg` utility is called with the `-d` flag, it deletes the entry for the specified layered product from the customer's `/usr/sys/conf/.product.list` file. In this case, the layered product is the `/dev/none` driver, represented by the `ESANONESTATIC100` product.

**15** Calls the `sysconfigdb` utility with the `-d` flag, which deletes the static `/dev/none` device driver from the customer's `/etc/sysconfigtab` database.

**16** Displays an appropriate message on the console terminal after the entry for the device driver product is removed from the customer's `/usr/sys/conf/.product.list` file.

**17** If the user chooses menu item 2 (delete a dynamically configured driver), calls the `sysconfig` utility with the `-u` flag to unconfigure the dynamically configured `/dev/none` device driver from the running system.

**18** Calls the `sysconfigdb` utility with the `-d` flag to delete the dynamically configured `/dev/none` device driver from the customer's `/etc/sysconfigtab` database.

## 3.6 Foreign Device Subset Control Program

A foreign device driver must be statically configured. Therefore, the subset control program for this driver does not need to provide a menu of configuration options. However, the subset control program must support the Remote Installation Services (RIS) utility, as this is the mechanism for installing foreign devices.

The RIS utility provides the ability to install foreign device driver product kits into a RIS area for subsequent installation on a client system. The RIS installation procedure bypasses `setld`'s `C INSTALL` phase, where the subset control program would ordinarily configure the driver into the system. Instead, the RIS installation procedure calls the subset control program at two times, passing either `EXTRACT` or `INSTALL` as an argument. These two phases are defined by the RIS utility, not by the `setld` utility. The subset control program must set the `ACT` environment variable in this situation.

The subset control program needs to contain an `EXTRACT` action phase for RIS. This phase invokes the subset control program at the end of the extract procedure when installing the kit on a RIS server. During this phase, the subset control program invokes the `kreg` utility to register the new product with the kernel, so that the driver is properly installed when the RIS server builds a new install kernel. In addition, the subset control program invokes the sysconfigdb utility to modify the /etc/sysconfigtab database. The modifications to the server by kreg and sysconfigdb occur only in the RIS area, as opposed to the server's root directory.

The /etc/sysconfigtab database created during this phase of the subset control program is copied onto the client system as part of the installation. This copy replaces the /etc/sysconfigtab database installed as part of the base operating system subset load. This ensures that the proper support for the driver exists on the system.

The subset control program is called with the INSTALL argument the first time the newly installed system boots up. At this time, the subset control program calls the kreg utility to register the driver with the new system. This adds the driver into the kernel that is built as part of the installation process. No modification to the /etc/sysconfigtab database is required at this point because the installation process has taken care of the required modification as part of the installation, during the EXTRACT phase.

A foreign device driver product may also be installed as a kernel product. If so, the subset control program must contain code to support both types of installation, as shown in Example 3-4.

### Example 3-4: Subset Control Program for the /dev/edgd Driver

```
#!/sbin/sh
#
#
#  EDGD.scp - Install files associated with the statically
#             configured /dev/edgd device driver product.
#
#

#
# RIS server installation of a foriegn kit
#
# In the case of RIS extract, the variable ACT is NULL, and
# the first parameter passed to the SCP specifies the phase.
#
[ "$ACT" ] ||  1
    ACT=$1

case "$ACT" in

#
# Configuration install phase takes place after the susbsets
# are loaded.  This phase configures the device driver into
# the system.  It is invoked on all installations of the kit:
# during CD-ROM or RIS-client Digital UNIX installation, and
# setld -l on an installed system.
#
C)
    case $1 in
```

**Example 3-4: (continued)**

```
    INSTALL) 2
        echo "INSTALL phase "

        # Register the files associated with the static
        # /dev/edgd device driver product.
        kreg -l EasyDriverInc EDGSTATIC100 /usr/opt/EDG100 3

        # Add the files associated with the static /dev/edgd
        # device driver product to the customer’s sysconfigtab
        # database.
        sysconfigdb -a -f /usr/opt/EDG100/sysconfigtab edgd 4

        # Reminder 5
        echo ""
        echo "The /dev/edgd device driver is installed on your system."
        echo "Before you can make use of the driver, you"
        echo "must build a new kernel by running doconfig."
        echo ""
        echo "See /usr/opt/EDG100/README for complete"
        echo "installation instructions"
        echo ""
      ;;
    esac
    ;;

#
# RIS Server kit installation phase
#
EXTRACT)  6

    echo "EXTRACT phase "

    #
    # The RIS server does this with ROOT set to the RIS area,
    # and the RIS area must be extracted, not linked to a CD-ROM.
    #

    # Register the files associated with the static /dev/edgd
    # device driver product.
    kreg -l EasyDriverInc EDGSTATIC100 /usr/opt/EDG100 7

    # Break link between /etc/sysconfigtab and /etc/.new..sysconfigtab
    # so we can run sysconfigdb for the RIS installation
    rm /etc/sysconfigtab  8

    # Copy the /etc/sysconfigtab database to client system.
    cp /etc/.new..sysconfigtab /etc/sysconfigtab 9

    # Add the files associated with the loadable /dev/edgd device
    # driver product to the customer’s sysconfigtab data base.
    sysconfigdb -a -f /usr/opt/EDG100/sysconfigtab edgd 10
    ;;
```

**Example 3-4: (continued)**

```
#
#  This phase is executed on a setld -d command which
#  is used to remove the subset from the system.
#
POST_D)  11

    kreg -d EDGSTATIC100    12
    rm -rf /usr/opt/EDG100  13
    sysconfigdb -d  edgd    14

    echo "The /dev/edgd device driver is no longer on the system." 15
    echo "Remember to build a new kernel to remove the /dev/edgd"
    echo "driver functionality."
    ;;

esac
exit 0
```

1 If ACT is null, sets the environment variable to the value of the first argument. The ACT environment variable is null during the RIS extract phase, when the RIS utility calls the subset control program with an argument of EXTRACT.

2 Handles the C INSTALL phase when the device driver is configured into the system.

3 Registers the driver files with the kernel, using the kreg utility. The kreg utility registers a device driver product by creating the /usr/sys/conf/.product.list file on the customer's system. This file contains registration information associated with the static device driver product.

   The SCP calls kreg with the following arguments:

   – The –l flag

     This flag indicates that the subset was loaded, and it directs kreg to register the device driver product as a new kernel extension.

   – Company name

     The company name is EasyDriverInc. The kreg utility places this name in the company name field of the customer's /usr/sys/conf/.products.list file.

   – Software subset name

     The software subset name for this device driver product is EDGSTATIC100. The subset name consists of the product code, subset mnemonic, and 3-digit version code. The kreg utility extracts information from the specified subset data and loads it into the customer's /usr/sys/conf/.products.list file.

– Directory name

The directory on the customer's system where kreg copies the files associated with this driver product is /usr/opt/EDG100. The kreg utility places this directory in the driver files path field of the customer's /usr/sys/conf/.products.list file.

4 Adds the sysconfigtab file fragment for the statically configured driver to the system's /etc/sysconfigtab database by calling the sysconfigdb utility with the following arguments:

– The –a flag

This flag causes sysconfigdb to add the device driver entry to the customer's /etc/sysconfigtab database.

– The –f flag

This flag precedes the name of the sysconfigtab file fragment whose device driver entry is to be added to the /etc/sysconfigtab database. This flag is used with the –a flag.

– The sysconfigtab file fragment

The kit developer at EasyDriver, Inc. specifies the path /usr/opt/EDG100/sysconfigtab to indicate the location of the sysconfigtab file fragment for the /dev/none device driver.

– Device driver name

The kit developer at EasyDriver, Inc. specifies none as the name of the driver whose associated information is added to the /etc/sysconfigtab database. This name is obtained from the Entry_Name attribute of the sysconfigtab file fragment, as described in *Writing Device Drivers: Tutorial*.

5 Displays a message indicating that the dirver is successfully installed on the system.

6 Handles installation of the kit on the RIS server.

7 Registers the driver files with the kernel, using the kreg utility. The kreg utility registers a device driver product by creating the /usr/sys/conf/.product.list file on the customer's system. This file contains registration information associated with the static device driver product.

8 Breaks the link to the /etc/sysconfigtab database on the client's system.

9 Copies the /etc/sysconfigtab database from the RIS area on the server to the client system.

10 Adds the sysconfigtab file fragment for the statically configured driver to the system's /etc/sysconfigtab database by calling the sysconfigdb utility with the –a and –f flags.

11 Handles the POST_D phase, when setld deletes the product subsets from the system.

12 Calls the kreg utility with the –d flag to deregister the driver with the kernel.

13 Removes the files from the product directory, EDG100, and any of its subdirectories.

14 Calls the sysconfigdb utility with the –d flag to delete the device driver from the customer's /etc/sysconfigtab database.

15 Displays a message on the console terminal, informing the user that the foreign device driver is no longer available on the system.

# Creating Subsets 4

A subset is the smallest installable entity in a kit that is compatible with the `setld` utility. It is up to you, the kit developer, to specify how many subsets your kit has and what files each contains. A good practice is to group files by related function or interdependence. For example, message libraries should be together in a subset to allow for easy localization and translation to other languages.

You specify the subsets and the files that each subset contains in a master inventory file. It is also up to you to specify the product attributes, such as the product name and version, and the subset definitions. You create a key file for this purpose. With these two files in place — the master inventory and key file — you can run the `kits` utility to create the subsets and control files. You perform the same steps when creating subsets for user products, kernel products, and foreign device products.

This chapter describes how to create the master inventory and key files, and how to use the `kits` utility to create subsets and subset control files. The ODB product is used as an example throughout the chapter to illustrate these tasks.

## 4.1 Creating the Master Inventory File

You can create a master inventory file with any text editor you like, or you can create the file with the `touch` command. The master inventory file name must consist of the product code and version, with the letters `mi` as a suffix. For example, you can create the master inventory for the ODB kit as follows:

```
% cd data
% touch OAT100.mi
```

The first time you process a kit, the master inventory file is empty. You must enter one record for each file that belongs on the kit. To get an initial list of these files, you can use the `newinv` command. Specify the file name of the empty master inventory file and the pathname of the source hierarchy's top-level directory. For example:

```
% newinv OAT100.mi ../src
```

This command invokes `newinv` on the master inventory file for the ODB product. It specifies the pathname to the source hierarchy as a relative path from the current the directory (`data`).

The `newinv` utility produces a list of files that are present in the source hierarchy and places you in the `vi` editor, or the editor specified by your `EDITOR` environment variable, so that you can make the required changes.

### Cautions

- Use extreme care when editing the master inventory file; fields in this file must be separated by single tab characters, not by spaces.
- The files listed in the master inventory file are given dot-relative pathnames. The `setld` utility usually works from the system's root directory, but the user can specify an alternative root directory with the `-D` option. For this reason, you should not use absolute pathnames in the master inventory file.

The master inventory file contains one record for each file in the kit. Each record in the master inventory file consists of three fields, described in Table 4-1.

**Table 4-1: Fields in Master Inventory Records**

| Field | Description |
|---|---|
| Flags | A 16-bit unsigned integer.<br>Bit 1 is the `v` (volatility) bit. When set, changes to the existing copy of the file can occur during kit installation. It is usually set for files such as `usr/spool/mqueue/syslog`.<br>Bit 2 is the `l` (link) bit. When set, the `STL_LinkCreate()` routine invoked in the subset control program (`.scp`) creates a forward link from the traditional system directories to the layered product `opt` areas.<br>The remaining bits are reserved; possible values for this field are therefore 0, 2, 4, or 6. |
| Pathname | The dot-relative (`./`) pathname of the file. |

**Table 4-1: (continued)**

| Field | Description |
|---|---|
| Subset identifier | The name of the subset containing the file. Subset names consist of the product code, subset mnemonic, and version number.<br><br>You must not include standard system directories in your subsets. In the ODB master inventory file, several records specify directories that are part of the standard system hierarchy. Instead of a subset identifier, these records specify RESERVED; this keyword prevents `setld` from attempting to overwrite existing directories. |

Example 4-1 shows that the ODB kit has two subsets. The OATODB100 subset contains utilities and libraries and must be installed if the product is to be used. The OATODBDOC100 subset contains the product's documentation and is not required to make the product work.

### Example 4-1: Master Inventory File for the ODB Kit

```
0       .       RESERVED
0       ./usr/opt       RESERVED
0       ./usr/opt/OAT100/OATODBDOC.Links        OATODBDOC100
0       ./usr/opt/OAT100/bin    OATODB100
4       ./usr/opt/OAT100/bin/attr       OATODB100
4       ./usr/opt/OAT100/bin/dcb.spr    OATODB100
4       ./usr/opt/OAT100/bin/dcb_defaults       OATODB100
4       ./usr/opt/OAT100/bin/dcb_diag.sed       OATODB100
4       ./usr/opt/OAT100/bin/docbld     OATODB100
4       ./usr/opt/OAT100/bin/unstamp    OATODB100
0       ./usr/opt/OAT100/lib    OATODB100
0       ./usr/opt/OAT100/lib/br OATODB100
4       ./usr/opt/OAT100/lib/br/README.dcb      OATODB100
4       ./usr/opt/OAT100/lib/br/attr.1  OATODBDOC100
4       ./usr/opt/OAT100/lib/br/dcb.ps  OATODBDOC100
4       ./usr/opt/OAT100/lib/br/docbld.1        OATODBDOC100
4       ./usr/opt/OAT100/lib/br/unstamp.1       OATODBDOC100
0       ./usr/opt/OAT100/lib/doclib     OATODB100
0       ./usr/opt/OAT100/lib/doclib/templates   OATODB100
4       ./usr/opt/OAT100/lib/doclib/templates/conv.braces       OATODB100
   .
   .
   .
```

For subsequent updates to the kit, use the existing version of the master inventory file for the input file. The `newinv` utility performs these additional steps:

- Creates a backup file, *inventory-file*.bkp
- Finds all the file and directory names in the source hierarchy

- Produces the following sorted groups of records:
  – Records containing pathnames only, representing files now present that were not in the previous inventory
  – Records representing files now present that were also present in the previous inventory (this list is empty the first time you create the inventory)
  – Records that were in the previous inventory but are no longer present (also empty the first time you create the inventory)
- Lets you edit the third of these groups, deleting records for files which no longer belong in the kit
- Lets you edit the group of new records by adding the flag and subset fields (see Table 4-1)
- Merges the three groups of records and sorts the result to produce a finished master inventory file that matches the source hierarchy

## 4.2 Key File

The key file identifies the product that the kit represents, such as its name, version number, and the name of the master inventory file for the kit. You create this file in the data directory with any text editor that you like. Its name must consist of the product code and version, with the letter `k` as a suffix. For example, `OAT100.k` is the key file for the ODB kit. Example 4-2 illustrates this key file.

**Example 4-2: Key File for the ODB Kit**

```
#       Product-level attributes
#
NAME='Orpheus Authoring Tools'
CODE=OAT
VERS=100
MI=OAT100.mi
COMPRESS=1
#
#       Subset definitions
#
```

**Example 4-2: (continued)**

```
%%
OATODB100       .       0       'Document Building Tools'
OATODBDOC100    .       2       'Document Tools Documentation'
```

As shown in the example, the key file is divided into the following two sections separated by a line containing two percent signs (%%):

- The product attributes section describes the naming conventions for the kit and provides kit-level instructions for the `kits` command. This section of the key file consists of several lines of attribute-value pairs. Table 4-2 describes the possible attribute-value pairs. Each attribute name is separated from its value by an equal sign (=). You can include comment lines, which begin with a number sign (#).
- The subset descriptor section describes each of the subsets in the kit and provides subset-level instructions for the `kits` command. This section contains one line for each subset in the kit. Each line consists of four fields separated by tab characters. You cannot include comments in this section of the key file. Table 4-3 describes the subset descriptor fields. In Example 4-2, the OATODB100 subset is mandatory; OATODBDOC100, which contains the documentation, is optional.

**Table 4-2: Key File Attributes Section**

| Attribute | Description |
|---|---|
| NAME | The product name; for example, Orpheus Authoring Tools |
| | Enclose the product name in single quotation marks (') if it contains any spaces. |
| CODE | A unique product code consisting of three numbers or uppercase letters, for example, OAT; the first character must be a letter |
| | The following codes are reserved by Digital: |
| | DNP, DNU, EPI, FOR, LSP, ORT, OSF, SNA, UDT, UDW, UDX, ULC, ULT, ULX, UWS |
| | Internal Digital product developers should contact the Software New Products Committee (SNPC) through product management to register a unique code. |
| | Third party developers should contact their Digital representatives to register a unique product code. |

**Table 4-2: (continued)**

| Attribute | Description |
|---|---|
| `VERS` | A 3-digit version code; for example, 100<br>The `setld` utility interprets this version code as 1.0.0. The first digit should reflect the product's major release number, the second the minor release number, and the third the upgrade level, if any. |
| `MI` | The name of the product's master inventory file consisting of the product name, code, and version plus the `mi` extension.<br>You create and maintain the master inventory file with the `newinv` utility. |
| `ROOT` | Not illustrated in the example, this optional attribute is reserved by Digital for the base operating system. For that use, it has a string value that names the root image file. Do not assign the `ROOT` attribute for a layered product. |
| `COMPRESS` | An optional flag that is set to 1 if you want to create compressed subset files. For kits in Direct CD–ROM (DCD) format, you must set this flag to zero (0).<br>Compressed files require less space on the distribution media (sometimes as little as 40% of the space required by uncompressed files), but they take longer to install than uncompressed files. If missing, this flag defaults to zero (`0`). |

**Table 4-3: Key File Subset Descriptor Fields**

| Field | Description |
|---|---|
| Subset identifier | A character string up to 80 characters in length, composed of the product code (for example, OAT), a mnemonic identifying the subset (for example, ODB), and the 3-digit version code (for example, 100). All letters in the subset identifier must be uppercase. |
| Reserved | Must be a single period ( `.` ). |

**Table 4-3: (continued)**

| Field | Description |
|---|---|
| Flags | A 16-bit unsigned integer<br>Digital reserves the use of the lower 8 bits.<br>You set bit 0, the sticky bit, to indicate that the subset cannot be removed.<br>You set bit 1 to indicate that the subset is optional.<br>The remaining bits are reserved; possible values for this field are therefore, 0, 2. |
| Subset description | A short description of the subset, delimited by single quotation marks ( ' ); for example,<br>`'Document Building Tools'`<br>**Note**<br>The percent sign character (%) is reserved in this field and must not be used for layered products. |

## 4.3 Running the kits Utility

After you create the master inventory and key files, you create subsets by running the `kits` command from the data directory. This command requires three arguments:

- Key file name
- Pathname for the source hierarchy
- Pathname for the output directory

For example, the ODB kit is built with the following command:

```
% cd /dcb_tools/data
% kits OAT100.k ../src ../output
```

The `kits` utility performs the following steps and reports its progress:

1. Creates the subsets.
2. Compresses each subset, if you specify the COMPRESS attribute in the key file.
3. Creates the installation control files listed in Table 4-4 and places them in the `instctrl` directory.

4. Creates the `INSTCTRL` file containing a `tar` image of all the installation control files. This file is placed in the output directory.

The subset files and the `INSTCTRL` file are constituents of the final kit.

**Table 4-4: Installation Control Files in the instctrl Directory**

| File | Description |
|---|---|
| *product-id*.comp | Compression flag file<br>This empty file is created only if you specified the COMPRESS attribute in the key file. Its presence signals to `setld` that the subset files are compressed. The ODB kit's compression flag file is named `OAT100.comp`. |
| *product-code*.image | Image data file<br>This file contains size and checksum information for the subsets. |
| *subset-id*.ctrl | Subset control file<br>This file contains `setld` control information. There is one subset control file for each subset. |
| *subset-id*.inv | Subset inventory file<br>This file contains an inventory of the files in the subset. Each record describes one file. There is one subset inventory file for each subset. |
| *subset-id*.scp | Subset control program<br>If you created subset control programs for your kit, these files are copied from the `scps` directory to the `instctrl` directory. There is one subset control program for each subset; if you have not created a subset control program for a subset, `kits` creates a blank file. |

The following sections describe the contents of the installation control files in detail.

### 4.3.1 Image Data File

The `setld` utility uses the image data file to verify that the subset images it loads from the installation media are uncorrupted before the actual installation process begins. The image data file name consists of the product's unique three-letter name with the string `image` for a suffix. The image data file contains one record for each subset in the kit. The following example illustrates `OAT.image`, the image data file for the ODB kit:

```
15923    70 OATODB100
24305   400 OATODBDOC100
```

Table 4-5 describes the three fields in each record.

**Table 4-5: Image Data File Fields**

| Field | Description |
|---|---|
| Checksum | The modulo-65536 (16-bit) checksum of the subset file (after compression, if the file is compressed) |
| Size | The size of the subset file in kilobytes (after compression, if the file is compressed) |
| Subset identifier | The product code, subset mnemonic, and version number |

### 4.3.2 Subset Control Files

The `setld` utility uses the subset control files as a source of descriptive information about subsets. A control file for each subset contains the following information:

- Descriptive product name and subset identifier.
- Descriptive subset identifier.
- Space required by the subset in the `root` file system, specified in bytes.
- Amount of the `usr` file system required by the subset specified in bytes.
- Amount of the `var` file system required by the subset specified in bytes.
- Disk volume identification information, consisting of two colon-separated integers (the volume number of the disk containing the subset archive and the number of diskettes required to contain the subset archive).
- Tape volume number and the subset's location on the tape, consisting of two colon-separated integers (the volume number of the tape containing the subset archive and the file offset at which the subset archive begins).

  On tape volumes, the first three files are reserved for a bootable operating system image and are not used by `setld`. An offset of 0 (zero) indicates the fourth file on the tape. The fourth file is a `tar` archive named `INSTCTRL`, containing the kit's installation control files (listed in Table 4-4).
- Dependency list (reserved – see Table 4-3).
- Subset control flag bits.

The following example illustrates `OATODBDOC100.ctrl`, the control file for the ODB kit's OATODBDOC100 subset:

```
NAME='Orpheus Authoring Tools OATODBDOC100'
DESC='Document Tools Documentation'
ROOTSIZE=0
USRSIZE=522090
VARSIZE=0
NVOLS=1:2
MTLOC=1:1
DEPS="."
FLAGS=34816
```

### 4.3.3 Subset Inventory Files

The subset inventory file describes each file in the subset, listing its size, checksum, permissions, and other information. This information is generated by the `kits` command and reflects the exact state of the files as they were in the source hierarchy from which the kit was built. It is used by `setld` to duplicate that state, thus transferring an exact copy of the source hierarchy to the customer's system. Example 4-3 shows the inventory file, `OATODBDOC100.inv`, for the ODB kit's OATODBDOC100 subset. The backslashes (\) in this example indicate line continuation and are not present in the actual file.

**Example 4-3: Sample Subset Inventory File**

```
4       983     01851   1065    0       100644  3/21/91 100     f\
       ./usr/opt/OAT100/lib/br/attr.1  none    OATODBDOC100
4       424997  63356   1065    10      100644  4/15/91 100     f\
       ./usr/opt/OAT100/lib/br/dcb.ps  none    OATODBDOC100
4       7283    03448   1065    10      100644  4/15/91 100     f\
       ./usr/opt/OAT100/lib/br/docbld.1        none    OATODBDOC100
4       6911    37501   1065    0       100644  3/21/91 100     f\
       ./usr/opt/OAT100/lib/br/docbld.5        none    OATODBDOC100
4       985     41926   1065    0       100644  3/21/91 100     f\
       ./usr/opt/OAT100/lib/br/unstamp.1       none    OATODBDOC100
```

Each record of the inventory is composed of 12 fields separated by tab characters. Table 4-6 describes the contents of these fields.

**Table 4-6: Subset Inventory Field Descriptions**

| Field | Name | Description |
|---|---|---|
| 1 | Flags | A 16-bit unsigned integer. |
| | | Bit 1 is the `v` (volatility) bit; when it is set, changes to the existing copy of the file can occur during kit installation. It is usually set for files such as `usr/spool/mqueue/syslog`. |
| | | Bit 2 is the `l` (link) bit; when it is set, the `STL_LinkCreate()` function creates a forward link from the standard system directories to the layered product areas. |
| | | The remaining bits are reserved; possible values for this field are therefore 0 2, 4, or 6. |
| 2 | Size | The actual number of bytes in the file. |
| 3 | Checksum | The modulo-65536 (16-bit) checksum of the file. |
| 4 | uid | The user ID of the file's owner. |
| 5 | gid | The group ID of the file's owner. |
| 6 | Mode | The 6-digit octal representation of the file's mode. |
| 7 | Date | The file's last modification date. |
| 8 | Revision | The version code of the product that includes the file. |
| 9 | Type | A letter that describes the file: |
| | | `b` – Block device. |
| | | `c` – Character device. |
| | | `d` – Directory containing one or more files. |
| | | `f` – Regular file. For regular files with a link count greater than one, see file type `l`. |
| | | `l` – Hard link. Other files in the inventory have the same inode number. The first (in ASCII collating sequence) is listed in the referent field. |
| | | `p` – Named pipe (FIFO). |
| | | `s` – Symbolic link. |
| 10 | Pathname | The dot-relative (`./`) pathname of the file. |
| 11 | Referent | For file types `l` and `s`, the path to which the file is linked; for types `b` and `c`, the major and minor numbers of the device; for all other types, `none.` |

**Table 4-6: (continued)**

| Field | Name | Description |
|---|---|---|
| 12 | Subset identifier | The name of the subset containing the file. |

### 4.3.4 Compression Flag File

The `setld` utility uses the presence of the compression flag file (*`product-id`*`.comp`) to determine whether the subset files are compressed. The compression flag is an empty file whose name consists of the product code and the version number with the string `comp` as a suffix; for example, `OAT100.comp`.

# 5 Producing Distribution Media

After you have gathered product files into subsets, you can move the subsets onto the distribution media in one of the following formats:

- `tar` format

  In `tar` format, the product files belonging to the same subset are dumped to the distribution media as a single file. During installation, the `setld` utility uncompresses the files, then moves them onto the target system, preserving the files' original directory structure. Kits for user and kernel products should be in `tar` format.

- Direct CD–ROM (DCD) format

  In DCD format, the files are written to the distribution media as a UNIX file system. Subsets distributed in DCD format cannot be compressed. Kits for foreign device products must be in DCD format.

Kits can be distributed on tape, disk, or CD–ROM, as follows:

- Magnetic tape

  Kits for user and kernel products can be distributed on magnetic tape. You cannot distribute foreign device kits on magnetic tape because this media does not support DCD format. Use the `gentapes` utility to produce kits for magnetic tape media.

- Disk

  Kits for any type of layered product can be distributed on disk or CD–ROM in either `tar` or DCD format. Use the `gendisk` utility to produce kits for disk media.

- CD–ROM

  Kits for CD-ROM are first produced on disk, then moved onto the CD–ROM. See the instructions that come with your CD–ROM device for information on copying files onto a CD–ROM.

The `gentapes` and `gendisk` utilities refer to a database file called `/etc/kitcap`. This file contains information about the kits that can be built on the system. This chapter describes how to edit the `/etc/kitcap` file and use `gentapes` and `gendisk` to produce kits for each type of media.

## 5.1 Editing the /etc/kitcap File

The /etc/kitcap file is a database of kit descriptors. Each record contains a product code and the names of the directories, files, and subsets that make up the product kit. The gentapes and gendisk utilities use this file when creating distribution media.

Before you can build your kit, you must add a record to the /etc/kitcap file to describe your kit. Add the record to the file according to the following conventions:

- Separate fields with colons (:).
- Indicate a continuation line with a backslash (\) at the end of the line.
- Begin comment lines with a number sign (#); comment lines are ignored by gentapes and gendisk.
- Delimit comment fields within a kitcap record by an opening number sign (#) and a closing colon (:).

The contents of a kitcap record differ depending on whether you are going to produce tape or disk media. Therefore, you must add one record for each media type on which you plan to distribute your kit.

### 5.1.1 Tape Media kitcap Record Format

The kitcap record for tape media specifies the following information:

- Name of the product, which consists of the product code and version number specified in the CODE and VERS fields of the key file.
- A code indicating the media type, either TK for TK50 tapes or MT for 9-track magnetic tapes.
- Product description. This entry is usually taken from the NAME field of the key file.
- Name of the kit's output directory, where the gentapes utility can find the subsets.
- Three SPACE files, which are empty files used to ensure compatibility with operating system kits.

  You create this file in the output area of the kit directory structure, by issuing the following commands:

  ```
  # touch space
  # tar -cf SPACE space
  ```

- The instctrl directory, relative to the output directory specification.
- The names of the subsets that make up the kit.

For example, the following record would be added to the /etc/kitcap file to produce the ODB kit on TK50 tapes:

```
OAT100TK | Orpheus Authoring Tools: \
   /dcb_tools/output:SPACE:SPACE:SPACE: \
   instctrl:OATODB100:OATODBDOC100
```

The product name, OAT100, is the same name that appears in the key file. The product description, ("Orpheus Authoring Tools") also appears in the key file. The name of the output directory is specified as /dcb_tools/output, and three SPACE files are included for compatibility with operating system kits. The last line of the record contains the instctrl directory and the names of the two subsets that make up the kit — OATODB100 and OATODBDOC100.

### 5.1.2 Disk Media kitcap Record Format

The kitcap record for a disk specifies the following information:

- Name of the product, which consists of the product code and version number specified in the CODE and VERS fields of the key file.
- The code HD, indicating disk type.
- The partition on the disk media where the product should be placed. The partition is a letter between a and h. Partition c is used most often, as it exists on most UNIX file systems.
- Product description, which must use underscores (_) in place of spaces. This entry is usually taken from the NAME field of the key file.
- Name of the kit's output directory, where the gendisk utility can find the product subsets.
- The kit's instctrl directory, relative to the output directory specification.
- Names of the subsets that make up the kit.

The following example shows the kitcap record for the /dev/none driver:

```
ESA100HD:c:/: \
  EasyDriver_none_driver: \
  /easy/output:instctrl:ESANONE100
```

Based on the information supplied in this record, the gendisk utility places the kit on the c partition, in the / directory of the disk media. The kit output directory is named /easy/output, and subset control information is in the instctrl directory. The kit consists of one subset, named ESANONE100.

## 5.2 Building a Kit on Magnetic Tape

With the product subsets in the output area of the kit directory structure, use the `gentapes` utility to create the kit on magnetic tape. The syntax of the `gentapes` command is as follows:

**gentapes** [ **-w-v** ] [ *hostname***:** ] *product-code* *special*

The `–w` option specifies that `gentapes` writes the tape without verifying it; the `–v` option specifies that the command verifies a tape without writing it first. If you specify neither option, `gentapes` writes the tape, rewinds it, and verifies its contents.

The optional *`hostname`* argument is the name of a remote TCP/IP network machine that contains the `/etc/kitcap` file. The utility searches `/etc/kitcap` on the remote machine for the *`product-code`* and uses it for creating the media. The colon (:) is a required delimiter for TCP/IP networks, and space is permitted between the colon and the *`product-code`*. If you do not specify a *`hostname`*, `gentapes` looks on your own system. You can use NFS file sharing to mount the kit files remotely on a system with the required tape drive.

The *`product-code`* is a user-defined code that describes the product. It should match the product name specified in the kitcap record, which is usually a concatenation of the `NAME` and `VERS` fields of the key file.

The *`special`* argument is the name of the device special file for the device, such as `/dev/nrmt0h`.

For example, the following command produces a tape kit containing the ODB user product:

```
% gentapes OAT100 /dev/nrmt0h
```

## 5.3 Building a Kit on a Disk

With the product subsets in the output area of the kit directory structure, use the `gendisk` utility to create the kit on a disk.

**Note**

> The `gendisk` utility supports diskettes but does not support creation of a chained diskette kit. A kit written to diskette must fit on a single diskette or be packaged on separate diskettes.

The syntax of the `gendisk` command is as follows:

**gendisk** [ **–wv** ] [ **–d** ] [ *hostname***:** ] *product-code* *special*

The –w option specifies that gentapes writes the tape without verifying it; the –v option specifies that the command verifies a tape without writing it first. If you specify neither option, gentapes writes the tape, rewinds it, and verifies its contents.

The optional *hostname* argument is the name of a remote TCP/IP network machine that contains the /etc/kitcap file. The utility searches /etc/kitcap on the remote machine for the *product-code* and uses it for creating the media. The colon (:) is a required delimiter for TCP/IP networks, and space is permitted between the colon and the *product-code*. If you do not specify a *hostname*, gendisk looks on your own system. You can use NFS file sharing to mount the kit files remotely on a system with the required tape drive.

The *product-code* is a user-defined code that describes the product. It should match the product name specified in the kitcap record, which is usually a concatenation of the NAME and VERS fields of the key file.

The *special* argument is the name of the device special file for the device, such as /dev/rrz1a.

You can use gendisk to produce kits in either tar or DCD format, depending on whether you use the –d option.

### 5.3.1 Preparing a tar Formatted Kit

To prepare a kit on disk for a user or kernel product, you use the gendisk utility without the –d option. You specify the product name and the device special file name. For example, the following command creates a tar formatted kit for the /dev/none driver on the c partition of the disk named rz0:

```
% gendisk ESA100 /dev/rz0c
```

### 5.3.2 Using DCD Format

To create a kit for a foreign device product, you run gendisk with the –d option. The utility creates a kit in DCD format on the target disk partition specified in the /etc/kitcap entry, then transfers the kit to that partition. For example, the following command creates a kit in DCD format for the /etc/edgd device driver:

```
# gendisk -d EDG100 /dev/rz0c
```

After running gendisk, you need to perform the following additional steps:

1. Copy the *name*.kit files to the / directory of the media so that the osfboot utility can interpret the kit.

2. Create links for modules referenced by the `osfboot` utility's boot-link support.
3. Copy the `/etc/sysconfigtab` file to the kit media. This file contains configuration information regarding the devices of the kit. This information is read by `osfboot` and passed to the boot-linked `vmunix` image when it starts up.
4. Create a *`subset`*`.kk` file in the `instctrl` directory of the kit. The RIS utility uses this when installing the kit into a RIS area.

**Note**

When testing a DCD kit, be sure to reference the kit media at its mount point. For instance, if you decide to use a spare disk partition for creating a media master area, you must reference your kit to the mount point of the device.

# Installing a Kit 6

Digital UNIX provides the system administrator with several options for installing layered product kits. For example:

- Using `setld`, the system administrator may install a kit on a running system, in single-user or multi-user mode.
- Using `osfboot`, the system administrator may install a kit before the initial installation and boot of a system. Foreign devices are installed in this way.
- Using RIS support, the system administrator may integrate a foreign kernel kit into a RIS environment. The kit provides information for rebuilding the RIS environment to support RIS installation of foreign devices.

This chapter shows how to install a user product, kernel product, and foreign device product, and how to install a kit in a RIS environment.

## 6.1 Installing a User Product

To install a user product, the system administrator runs the `setld` utility. For example, the ODB product could be installed as follows, assuming it is distributed on CD–ROM:

1. Place the CD–ROM in its caddy and insert it into the driver.
2. As superuser or `root`, create a directory to be the mount point for the CD–ROM, such as `/mnt`:

   ```
   # mkdir /mnt
   ```

3. Mount the CD–ROM on `/mnt`. For example, if the CD–ROM device is located on the `c` partition of `rz4`, you would enter the following command:

   ```
   # mount -r /dev/rz4c /mnt
   ```

   After mounting the CD–ROM, you can change to the `/mnt` directory and view the directories on the CD–ROM.

4. Install the user product subsets:

   ```
   # setld -l /mnt/ALPHA/OAT100
   ```

   The `setld` utility displays prompts and messages to guide you through the installation process.

5. When the installation is complete, unmount the CD–ROM:

   ```
   # umount /mnt
   ```

See the *Installation Guide* for more information on using the `setld` utility to install layered products.

## 6.2 Installing a Kernel Product

To install a kernel product, the system administrator runs the `setld` utility, then reconfigures the kernel by running the `doconfig` utility. For example, the `/dev/none` driver could be installed as follows, assuming the kit is distributed on CD–ROM:

1. Place the CD–ROM in its caddy and insert it into the driver.
2. As superuser or `root`, create a directory to be the mount point for the CD–ROM, such as `/mnt`:

   ```
   # mkdir /mnt
   ```

3. Mount the CD–ROM on `/mnt`. For example, if the CD–ROM device is located on the `c` partition of `rz4`, you would enter the following command:

   ```
   # mount -r /dev/rz4c /mnt
   ```

4. Install the device driver:

   ```
   # setld -l /mnt/ALPHA/ESA100
   ```

   The `setld` utility displays prompts and messages to guide you through the installation process.

5. When the installation is complete, unmount the CD–ROM:

   ```
   # umount /mnt
   ```

   At this point, the kit is installed on the system, but the driver is not configured into the kernel. You must perform steps 5 – 8 to reconfigure the kernel.

6. Copy the current version of the kernel to another file, in case you need to

recover it should the reconfiguration fail:

```
# mv /vmunix /vmunix.save
```

7. Set your default dirctory to /usr/sys/conf and run the doconfig utility to configure the driver into the kernel:

```
# cd /usr/sys/conf
# doconfig
```

8. Copy the vmunix kernel from its current location to the / directory. The current location is the directory where the vmunix (the currently configured kernel) resides. This directory has the form /usr/sys/NAME/vmunix.

```
# cp vmunix /vmunix
```

9. Restart the system with the new kernel:

```
# /usr/sbin/shutdown –r now
```

When the system starts up, the /dev/none driver is available on the system.

See the *Installation Guide* for more information on using the setld utility to install layered products. See doconfig(8) for more information on the doconfig utility.

## 6.3 Installing a Foreign Device Product

The system administrator installs a foreign device kit with the osfboot utility, which runs from the console prompt. To follow this procedure, you need two kits: the CD–ROM containing the Digital UNIX kit and the kit containing the foreign device product.

For example, to install the /dev/edgd driver as a foreign device, run osfboot as follows:

1. Insert the Digital UNIX kit in the CD–ROM drive.
2. From the console prompt, invoke the osfboot utility with the fg flags, as follows:

```
>>> boot –fl fg dka400
```

The utility reads bootstrap code from the CD–ROM for the system you are installing.

3. The osfboot utility prompts you for the name of the device from which to install the foreign device. Enter the appropriate device name, for

example:

```
Enter Device Name: dva0
```

4. The utility prompts you for the name of the kit containing the foreign device. To install the /dev/edgd driver, you would enter the following:

```
Enter Kit Name: edgd.kit
```

5. Insert the media and press the return key:

```
Insert media for kit 'dva0:edgd.kit', hit <return> when
ready: Return
```

6. You may want to install more than one foreign device kit. Therefore, osfboot prompts you to enter more kit names. Press the return key to indicate that there are no more kits:

```
Enter Kit Name: Return
Insert boot media, hit <return> when ready: Return
```

7. After you have entered all the kit names, osfboot proceeds to link the modules into the kernel. When the link operation has completed, it prompts you to insert the media containing the foreign device kit, then prompts you to insert the media containing the base system:

```
Insert media for kit 'dva0:edgd.kit', hit <return> when
ready: Return
Insert boot media, hit <return> when ready: Return
```

## 6.4 Installing a Product into a RIS Area

A user product or kernel product kit can be installed in a RIS area on a RIS server by running the ris utility. When the product has been installed in the RIS area, RIS clients can install the software from that RIS area using the setld utility.

To install the product in the RIS area on the server, run the ris utility as follows:

1. Log onto the server as root and invoke the ris utility:

```
# /usr/sbin/ris
```

2. From the RIS main menu, choose the option INSTALL software

`products` by entering `i` at the prompt:

```
*** RIS Utility Main Menu ***

Choices without key letters are not available.

    a) ADD a client
    d) DELETE software products
    i) INSTALL software products
     ) LIST registered clients
     ) MODIFY a client
     ) REMOVE a client
    s) SHOW software products in remote installation environments
    x) EXIT
Enter your choice: i
```

3. The RIS Software Installation Menu is displayed. Enter the option 1, `Install software into a new area` or 2, `Add software into an existing area` at the prompt:

```
RIS Software Installation Menu:

    1) Install software into a new area
    2) Add software into an existing area
    3) Return to previous menu
Enter your choice:
```

   See *Sharing Software on a Local Area Network* for more information on installing software on the RIS area.

To install the product kit from the RIS server onto the client system, register the client system with the RIS server, then use the `setld` utility, as follows:

1. On the server, run the `ris` utility to register the client system by choosing option 2, `ADD a client` from the main menu:

```
# /usr/sbin/ris
#
*** RIS Utility Main Menu ***

Choices without key letters are not available.

    a)  ADD a client
    d)  DELETE software products
    i)  INSTALL software products
     )  LIST registered clients
     )  MODIFY a client
     )  REMOVE a client
    s)  SHOW software products in remote installation environments
    x)  EXIT
Enter your choice: a
```

2. Enter the client information requested by the prompts, as described in *Sharing Software on a Local Area Network*.

3. As superuser or `root` on the client system, install the device driver subset from the RIS area. For example, if the RIS area is on node `visier`, you would enter the following command:

   ```
   # setld –l visier:
   ```

   The `setld` utility displays prompts and messages to guide you through the installation process. See the *Installation Guide* for more information on using the `setld` utility to install layered products.

   If the product is a user product, the installation is complete. If the product is a kernel product, you must reconfigure the kernel using steps 4 – 7.

4. Copy the current version of the kernel to another file, in case you need to recover it should the reconfiguration fail.

   ```
   # mv /vmunix /vmunix.save
   ```

5. Set your default directory to `/usr/sys/conf` and run the `doconfig` utility to configure the driver into the kernel:

   ```
   # cd /usr/sys/conf
   # doconfig
   ```

6. Copy the `vmunix` kernel from its current location to the `/` directory:

   ```
   # cp vmunix /vmunix
   ```

7. Restart the system with the new kernel.

   ```
   # /usr/sbin/shutdown –r now
   ```

# Digital UNIX Standard Directory Hierarchy A

The important directories in the Digital UNIX standard hierarchy are shown in Figure A-1 and Figure A-2. Not all of the directories in the entire hierarchy are shown; the directories that are illustrated are the ones that you should use to ensure that your product is portable to other systems. Some of the illustrated directories are actually symbolic links.

**Figure A-1: Base System Directory Hierarchy**

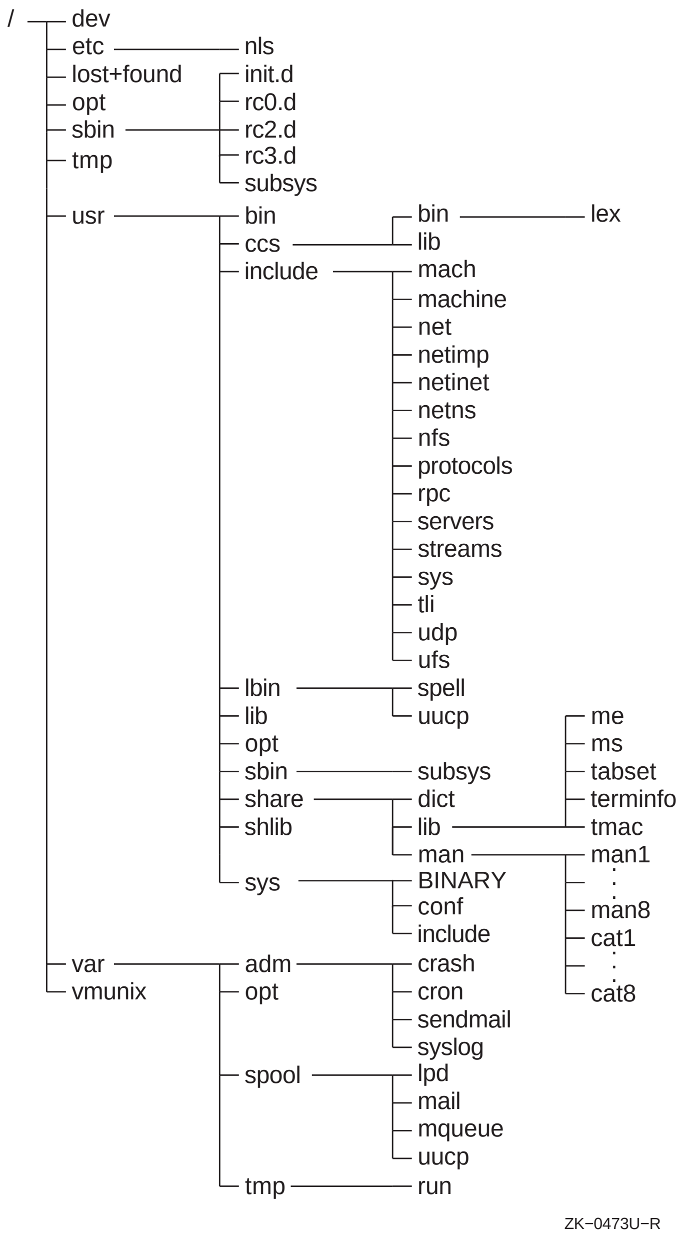


Table A-1 describes the contents and purpose of the directories shown in Figure A-1.

**Table A-1: Contents and Purpose of Base System Directories**

| Directory | Description |
|---|---|
| `/` | The root directory of the file system |
| `/dev/` | Block and character device files |
| `/etc/` | System configuration files and databases; nonexecutable files |
|   `nls/` | National language support databases |
| `/lost+found/` | Files located by `fsck` |
| `/opt/` | Optional application packages such as layered products |
| `/sbin/` | Commands essential to boot the system<br>Most of these commands depend on shared libraries or the loader and have other versions in `/usr/bin` or `/usr/sbin`. |
|   `init.d/` | System state rc files |
|   `rc0.d/` | The rc files executed for system-state 0 |
|   `rc2.d/` | The rc files executed for system-state 2 |
|   `rc3.d/` | The rc files executed for system-state 3 |
| `/subsys/` | Loadable kernel modules required in single-user mode |
| `/tmp/` | System-generated temporary files, usually not preserved across a system reboot. |
| `/usr/` | Most user utilities and applications |
|   `bin/` | Common utilities and applications |
|   `ccs/` | C compilation system; tools and libraries used to generate C programs |
|     `bin/` | Development binaries such as `cc`, `ld`, and `make` |
|     `lib/` | Development libraries and back ends |
|   `include/` | Program header (include) files; not all subdirectories are listed below |
|     `mach/` | Mach-specific C include files |
|     `machine/` | Machine-specific C include files |
|     `net/` | Miscellaneous network C include files |
|     `netinet/` | C include files for Internet standard protocols |
|     `netns/` | C include files for XNS standard protocols |
|     `nfs/` | C include files for Network File System |

**Table A-1: (continued)**

| Directory | Description |
|---|---|
| protocols/ | C include files for Berkeley service protocols |
| rpc/ | C include files for remote procedure calls |
| servers/ | C include files for servers |
| sys/ | System C include files (kernel data structures) |
| tli/ | C include files for Transport Layer Interface |
| ufs/ | C include files for UNIX File System |
| lbin/ | Back-end executable files |
| spell/ | Spell back-end |
| uucp/ | UNIX-to-UNIX Copy (UUCP) programs |
| lib/ | Consists entirely of links to libraries located elsewhere (/usr/ccs/lib), (/usr/libin), (/usr/share/lib), (/X11/lib); included for compatibility |
| opt/ | Optional application packages such as layered products |
| sbin/ | System administration utilities and system utilities |
| share/ | Architecture-independent ASCII text files |
| dict/ | Word lists |
| lib/ | Various libraries |
| me/ | Macros for use with the me macro package |
| ms/ | Macros for use with the ms macro package |
| tabset/ | Tab description files for a variety of terminals; used in /etc/termcap |
| terminfo/ | Terminal information database |
| tmac/ | Text processing macros |
| man/ | Online reference pages |
| man1/ | Source for user command reference pages |
| man2/ | Source for system call reference pages |
| man3/ | Source for library routine reference pages |
| man4/ | Source for file format reference pages |
| man5/ | Source for miscellaneous reference pages |
| man7/ | Source for device reference pages |
| man8/ | Source for administrator command reference pages |
| cat1-cat8 | Formatted versions of files in man1 – man8 |

**Table A-1: (continued)**

| Directory | Description |
|---|---|
| shlib/ | Binary loadable shared libraries; shared versions of libraries in /usr/ccs/lib |
| sys/ | System configuration files |
| BINARY | Object files |
| conf/ | Kernel configuration control files |
| include/ | Header files |
| /var/ | Multipurpose log, temporary, varying, and spool files |
| adm/ | Common administrative files and databases |
| crash/ | For saving kernel crash dumps |
| cron/ | Files used by cron |
| sendmail/ | Configuration and database files for sendmail |
| syslog/ | Files generated by syslog |
| opt/ | Optional application packages such as layered products |
| run/ | Files created when daemons are running |
| spool/ | Miscellaneous printer and mail system spooling directories |
| lpd/ | Line printer spooling directories |
| mail/ | Incoming mail messages |
| mqueue/ | Undelivered mail queue |
| uucp/ | UUCP spool directory |
| tmp/ | Application-generated temporary files that are kept between system reboots |
| subsys/ | Loadable kernel modules required in multi-user mode |
| /vmunix | Pure kernel executable (the operating system loaded into memory at boot time) |

### Figure A-2: X Directory Hierarchy

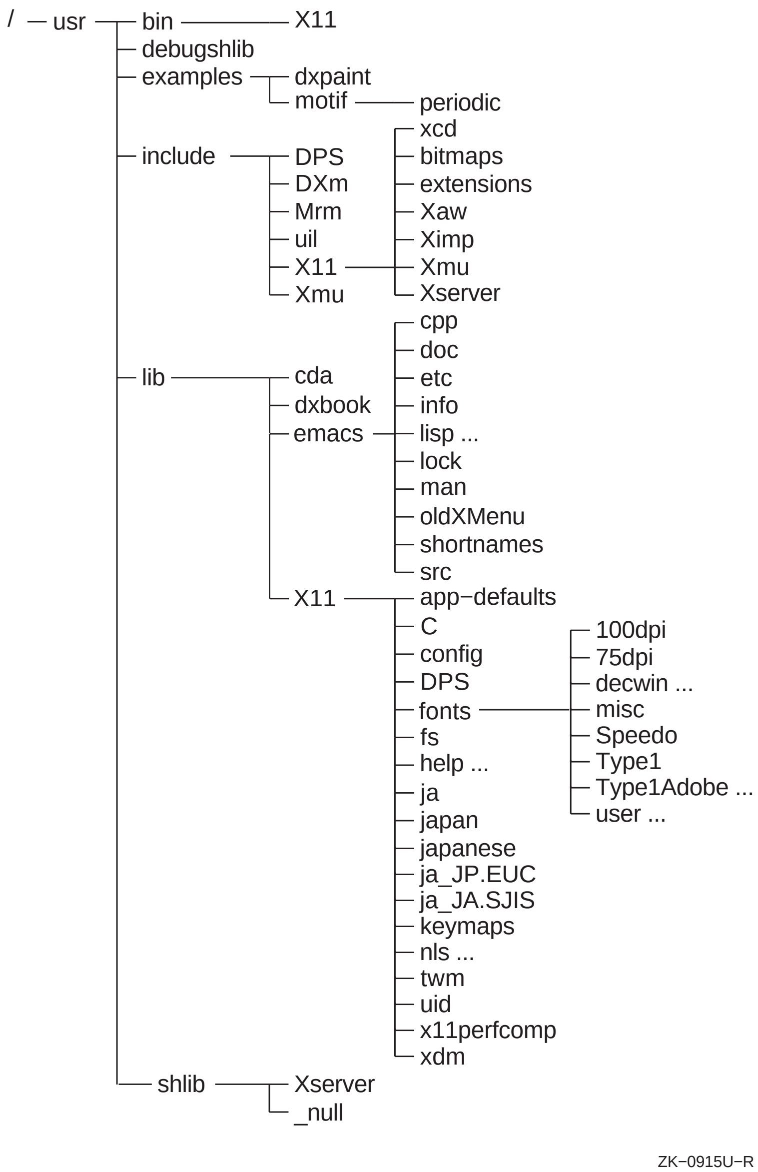


Table A-2 describes the contents and purpose of the directories shown in Figure A-2.

**Table A-2: Contents and Purpose of X Directories**

| Directory | Description |
|---|---|
| `/usr/` | Most user utilities and applications |
| `  bin/` | Common utilities and applications |
| `    X11/` | X applications |
| `      demos/` | Miscellaneous demo programs |
| `  examples/` | Example programs |
| `    dxpaint/` | Sample Paint image |
| `    motif/` | Motif example programs |
| `  include/` | Header files |
| `    DPS/` | Files for DPS |
| `    DXm/` | Files for libDXm |
| `    Mrm/` | Files for libMrm |
| `    uil/` | UIL header files |
| `    X11/` | X C header files |
| `      bitmaps/` | X bitmaps |
| `      extensions/` | Header files for use with X extensions |
| `      Xaw/` | Files for libXaw |
| `      Xserver/` | Header files used for loadable X server libraries |
| `    Xm/` | Header files for libXm |
| `  lib/` | Static archive X libraries |
| `    cda/` | CDA style guides |
| `    dxbook/` | Default Bookreader bookshelf |
| `    emacs/` | Emacs directory base |
| `    X11` | |
| `      app-defaults/` | System-wide resource files for X client applications |
| `      config/` | Imake config files |
| `      DPS/` | Display Postscript files |
| `      fonts/` | Font files |
| `        100dpi/` | 100 dpi fonts from X Consortium |
| `        75dpi/` | 75 dpi fonts from X Consortium |

**Table A-2: (continued)**

| Directory | Description |
|---|---|
| `decwin/` | DECwindows fonts |
| `100dpi/` | 100 dpi fonts |
| `75dpi/` | 75 dpi fonts |
| `misc/` | Fonts from X Consortium |
| `Speedo/` | Speedo scalable fonts |
| `Type1/` | Type1 scalable fonts |
| `Type1Adobe/` | Adobe Type1 scalable fonts |
| `afm/` | Adobe font metrics |
| `user` | Fonts from layered products and local installations |
| `100dpi/` | 100 dpi fonts |
| `75dpi/` | 75 dpi fonts |
| `misc/` | Other fonts |
| `fs/` | Fontserver config and error log files |
| `help/` | Help files for X client applications; subdirectories as applicable |
| `keymaps/` | Keymaps for various keyboards |
| `twm/` | Default configuration for `twm` window manager |
| `uid/` | User Interface Definitions for X client applications |
| `x11perfcomp/` | Scripts for analyzing `x11perf` output |
| `xdm/` | X Display Manager configuration and resource files, and error log |
| `bitmaps/` | Program-specific bitmaps |
| `appdata/` | Generic program-specific data |
| `shlib/` | Shareable libraries |
| `X11/` | Shareable libraries loaded by X server |

# Glossary

This glossary defines terms used in this manual.

attribute-value pair
: In a software kit's key file, a line specifying the name and value for a single attribute of the kit. Controls how the kit is built by the `kits` command and how it is installed by the `setld` utility.

collating symbol
: In a regular expression, a name that defines a particular subset of the available characters, such as lowercase characters, in a collating sequence that uses multicharacter strings to represent single characters.

dependency expression
: In a subset control program (SCP), a Backus-Naur form (postfix) logical expression consisting of subset identifiers and relational operators to describe the current subset's relationship to the named subsets. See also **subset dependency**.

kernel
: The kernel is a software entity that runs in supervisor mode and does not communicate with a device except through calls to a device driver.

Kernel framework
: See **subsystem**.

layered product
: In `setld` and product kit development, an optional software product designed to be installed as an added feature of the Digital UNIX system.

locking
: In software installation by the `setld` utility, the act of inserting a new subset's name in the lock file of an existing subset so that an attempt to remove the latter subset will flag the user with a dependency warning. In a version control system, the creation and use of information flagging a version control file as being checked out for editing.

name_data.c file
: The name_data.c file provides a convenient place to size the data structures and data structure arrays that device drivers use. In addition, the file can contain definitions that third-party device writers might want their customers to change. This file is particularly convenient for third-party driver writers who do not want to ship device driver sources.

Name.list file
: The `NAME.list` file is a copy of the `.product.list` file that is created when the system manager installs the device driver kit supplied by a third-party vendor.

setld
: The `setld` utility allows the transfer of the contents of the device driver kit to a customer's system on Digital UNIX

source hierarchy
: For building software kits, the directory tree and files that are to be compiled by the `kits` command into subsets for a kit.

subset control program (SCP)
: A subset control program (SCP) is a program written by the kit developer that contains path specifications for all of the files related to a given product kit. The SCP is invoked by `setld` during the installation of the kit.

subset
: The smallest installable component of a software kit for the `setld` utility. Contains files of any type, usually interrelated in some way.

subset dependency
: The condition in which a given subset requires the presence, or lack thereof, of other subsets in order to function properly. Evaluated by a subset's SCP under control of the `setld` utility. See also **dependency expression**.

target
: Also called a **target file**. In the `make` utility, an entity to be built from its **dependents**. An executable program is a target that is built from one or more object modules.

sysconfigdb
: The `sysconfigdb` utility is a system management tool that maintains the `sysconfigdb` database.

sysconfigtab database

The `sysconfigtab` database contains the information provided in the `sysconfigtab` file fragments. This information is appended to the `sysconfigtab` database when the system manager runs the `sysconfigdb` utility during the installation of a kit.

target hierarchy

For building software kits, the directory tree into which a software kit is placed by the `kits` command.

# Index

## U



## V

# How to Order Additional Documentation

## Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

## Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a 1200- or 2400-bps modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

## Telephone and Direct Mail Orders

| Your Location | Call | Contact |
|---|---|---|
| Continental USA, Alaska, or Hawaii | 800-DIGITAL | Digital Equipment Corporation<br>P.O. Box CS2008<br>Nashua, New Hampshire 03061 |
| Puerto Rico | 809-754-7575 | Local Digital subsidiary |
| Canada | 800-267-6215 | Digital Equipment of Canada<br>Attn: DECdirect Operations KAO2/2<br>P.O. Box 13000<br>100 Herzberg Road<br>Kanata, Ontario, Canada K2K 2A6 |
| International | ————— | Local Digital subsidiary or approved distributor |
| Internal[a] | ————— | SSB Order Processing – NQO/V19<br>*or*<br>U. S. Software Supply Business<br>Digital Equipment Corporation<br>10 Cotton Road<br>Nashua, NH 03063-1260 |

[a] For internal orders, you must submit an Internal Software Order Form (EN-01740-07).

# Reader's Comments

**Digital UNIX**
Guide to Preparing Product Kits

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: `readers_comment@zk3.dec.com`
- Fax: (603) 881-0120, Attn: UEG Publications, ZKO3-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

| **Please rate this manual:** | Excellent | Good | Fair | Poor |
|---|---|---|---|---|
| Accuracy (software works as manual says) | ☐ | ☐ | ☐ | ☐ |
| Completeness (enough information) | ☐ | ☐ | ☐ | ☐ |
| Clarity (easy to understand) | ☐ | ☐ | ☐ | ☐ |
| Organization (structure of subject matter) | ☐ | ☐ | ☐ | ☐ |
| Figures (useful) | ☐ | ☐ | ☐ | ☐ |
| Examples (useful) | ☐ | ☐ | ☐ | ☐ |
| Index (ability to find topic) | ☐ | ☐ | ☐ | ☐ |
| Usability (ability to access information quickly) | ☐ | ☐ | ☐ | ☐ |

**Please list errors you have found in this manual:**

Page Description

________ ____________________________________________

________ ____________________________________________

________ ____________________________________________

________ ____________________________________________

________ ____________________________________________

**Additional comments or suggestions to improve this manual:**

____________________________________________________

____________________________________________________

____________________________________________________

____________________________________________________

____________________________________________________

**What version of the software described by this manual are you using?** ____________

Name/Title ______________________ Dept. ____________

Company ______________________ Date ____________

Mailing Address ____________________________________

______________ Email ______________ Phone ____________

Do Not Cut or Tear − Fold Here and Tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

**BUSINESS REPLY MAIL**
FIRST−CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UEG PUBLICATIONS MANAGER
ZKO3−3/Y32
110 SPIT BROOK RD
NASHUA NH 03062−9987

Do Not Cut or Tear − Fold Here

Cut on Dotted Line