ValidPACKAGER™ REFERENCE MANUAL

Manual Number: MN223 Rev A

10 March 1986

Valid Logic Systems, Incorporated
2820 Orchard Parkway
San Jose, California 95134
(408) 945-9400 Telex 371 9004

NEW PACKAGER FEATURES

This manual describes version 7.28 of the Packager program. You may be using version 7.25. When you run the Packager, the first line of your listing file (PSTLST) will show the version of the Packager:

Valid Logic Systems, Inc. Packager 7.25:1June85

or

Valid Logic Systems, Inc. Packager 7.28:1June85

If you are using version 7.25 of the Packager program, then several of the features described in this manual will not be available to you. These include the following:

- The GROUP property.
- The subdirectory (-s) option of the "package" command.
- These directives:

  FREE_GROUPING
  HARD_GROUPING
  HARD_LOC_SEC
  LIBRARY
  PART_TYPE_LENGTH
  PRINT_PIN_LIST
  USE_PIN_GROUP
  DOCUMENT_ERRORS

- The following output files:

  PSTPCHG (physical changes file)
  PSTBCHG (binding changes file)

- The subtype suffix specification for physical part tables.

- The verbose (-v) option for feedback diagnostic messages.

- Feedback: in the 7.25 Packager, if you don't list all pins for a section, the Packager ignores that section and doesn't do feedback on it. In the 7.28 Packager, if a pin isn't listed, the Packager assumes that the pin is not connected (i.e., it is connected to the 'NC' net).

- Compact PIN_NUMBER syntax specification (see Appendix A).

TABLE OF CONTENTS

SECTION 1
OVERVIEW

1.1 HOW TO USE THIS MANUAL

This manual is a reference guide for the Packager. Sections 1 through 4 contain an overview of the Packager, an introduction to running the Packager, and some basic information on specific functions and concepts necessary to understand the Packager. Sections 5 through 11 describe important Packager features and cover important concepts in greater depth and detail. Sections 12 through 15 are strictly reference material: output file formats, error messages, directives, and a glossary.

Everyone should read sections 1 through 4. Most users will also need to read portions of Sections 5 through 11. The manual has a ''top-down'' design; the first few sections provide a global view of the Packager and contain forward references to later sections that describe particular functions in more detail.

1.2 INTRODUCTION TO THE PACKAGER

After a logical design is completed, it must be translated into a physical design. The logical design must be ''packaged'' into physical chips on a printed circuit board. The Packager is a bridge between the logical and physical designs, and information flows in both directions across this bridge. The Packager processes the compiled logic design and produces files for a physical design system. The Packager also accepts feedback files from a physical design system (to modify the Packager's original physical design) and produces a back annotation file that is used by the Graphics Editor to incorporate physical part numbers and physical pin numbers into the original logical design. During the packaging process, the Packager creates ''state files'' to maintain a consistent physical design for successive packaging runs. Figure 1 shows an overview of the Packager's relation to the Graphics Editor, the Compiler, and a physical design system.

The Packager performs the following functions:

- Logical to Physical Assignments
- Expansion of Structured Parts
- Removing Phantom Wire-OR and Wire-AND Bodies
- Checking Loading Constraints and Unconnected Signals
- Processing Feedback Files
- Creating Net and Parts Lists
- Preparing the Back Annotation File

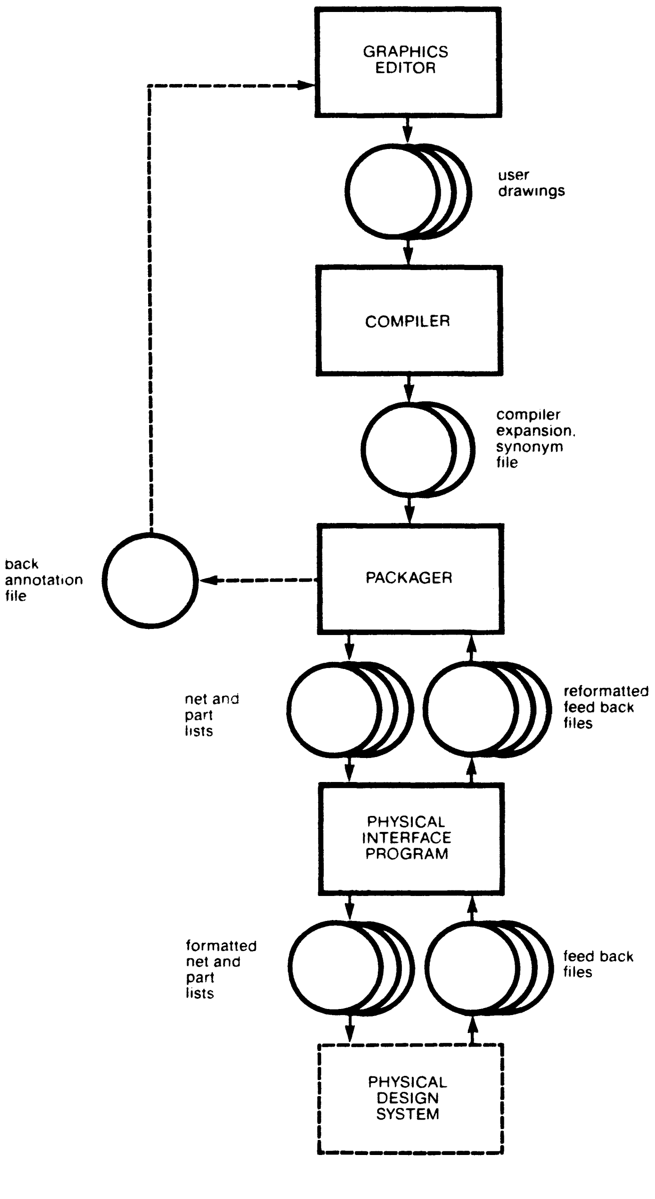

Figure 1-1. Overview of Packager and Related Systems

LOGICAL TO PHYSICAL ASSIGNMENTS

Each logical part, signal, and pin is given a physical name by the Packager. (These names can be changed by the user.)

EXPANSION OF STRUCTURED PARTS

Parts with a SIZE property are expanded into the corresponding number of parts, and the vectored signals connected to the ''sized'' part are assigned to the expanded parts. The Packager also creates new versions of parts with TIMES properties.

REMOVING PHANTOM WIRE-OR AND WIRE-AND BODIES

All wire-OR and wire-AND bodies in the design are replaced with explicit wire ties.

NET AND LOAD CHECKS

The Packager checks each net to make sure that the loading specifications of the parts are not violated. The Packager also checks for unconnected signals on nets.

PROCESSING FEEDBACK FILES

The Packager processes feedback files from the physical design system to modify its original physical design.

CREATING NET AND PARTS LISTS

The Packager creates a net list and part list which contain the information to be passed to a physical design system.

PREPARING THE BACK ANNOTATION FILE

The Packager creates a back annotation file for the Graphics Editor.

SECTION 2
RUNNING THE PACKAGER

This section describes how to run the Packager and discusses the input and output files as well as the overall data flow. The Packager is an unusual program in that some of its output files are used as inputs to the next Packager run for the design. Thus, the initial Packager run differs from later runs.

2.1 INITIAL PACKAGER RUN

Figure 2 shows an overview of the input and output files for an initial run of the Packager. The files CMPEXP and CMPSYN are the expansion and synonym files from a Compiler run for logic. These files contain the logical design description. The file packager.cmd is a text file maintained by the user that contains directives (commands) from the user to the Packager. These commands control specific Packager functions. The library files and physical part tables contain physical information about the parts included in the logical design (e.g., pin numbers, number of sections per physical part, and electrical characteristics of the part). The Packager processes information from these input files and produces more than a dozen output files.

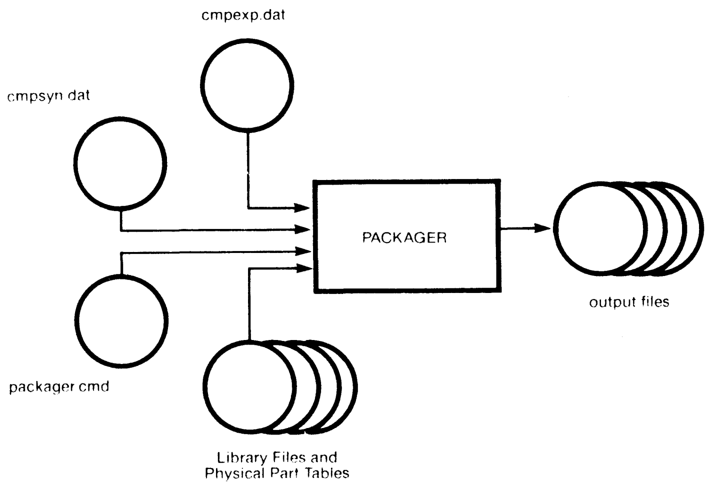

Figure 2-1. Overview of Initial Packager Run

Before running the Packager, you must compile your design for logic and prepare a Packager directives file. To run the Packager, enter this command after the prompt:

package

When you enter this command, the Packager uses the current Compiler output files, the Packager directives file (packager.cmd), and the library files and physical part tables specified in the directives file. (The "package" command can also be used in the form "package [-s] [*root drawing name*]". This enables you to package different designs in the same directory but still maintain the design files separately. This option is discussed in more detail at the end of this section.)

While it is running, the Packager sends messages to the terminal reporting on the progress of activity. When the Packager finishes, the terminal shows the number of errors (if any) and CPU time. Here is an example of a simple

Packager directives file:

```
library_file '/u0/lib/lsttl/lsttl.prt';
warnings on;
oversights on;
end.
```

The first three lines of the example file are directives. Each directive ends with a semicolon (;). The last line of the file is always "END." Directives allow you to specify certain choices for the Packager. For example, WARNINGS ON tells the Packager to print warning messages. Most directives are optional. If you omit them, the Packager uses a preset choice called a default. However, the LIBRARY_FILE or LIBRARY directive is required to specify the library file(s) containing the physical information for the parts in your design. Some of the more commonly-used directives are discussed later in this section. Section 12 contains a complete description of all Packager directives. The directives are listed in that section in alphabetical order, with syntax and purpose fully defined.

> **NOTE:** The Packager directives file and the Compiler output files must be in your current directory when you enter the command "package". Libraries can be in a different directory, since the LIBRARY_FILE directive includes the full path name for the library file.

Figure 3 shows the input and output files from a first run of the Packager. All Packager output files have names beginning with the letters "pst" which stands for "post processing." You are now "post processing" your design for use with another system. This manual refers to output files by their logical name. The actual name depends on the particular operating system. The UNIX file name is the logical file name in lower case followed by the extension ".dat". For example, PSTLST becomes pstlst.dat. The output files are described below.

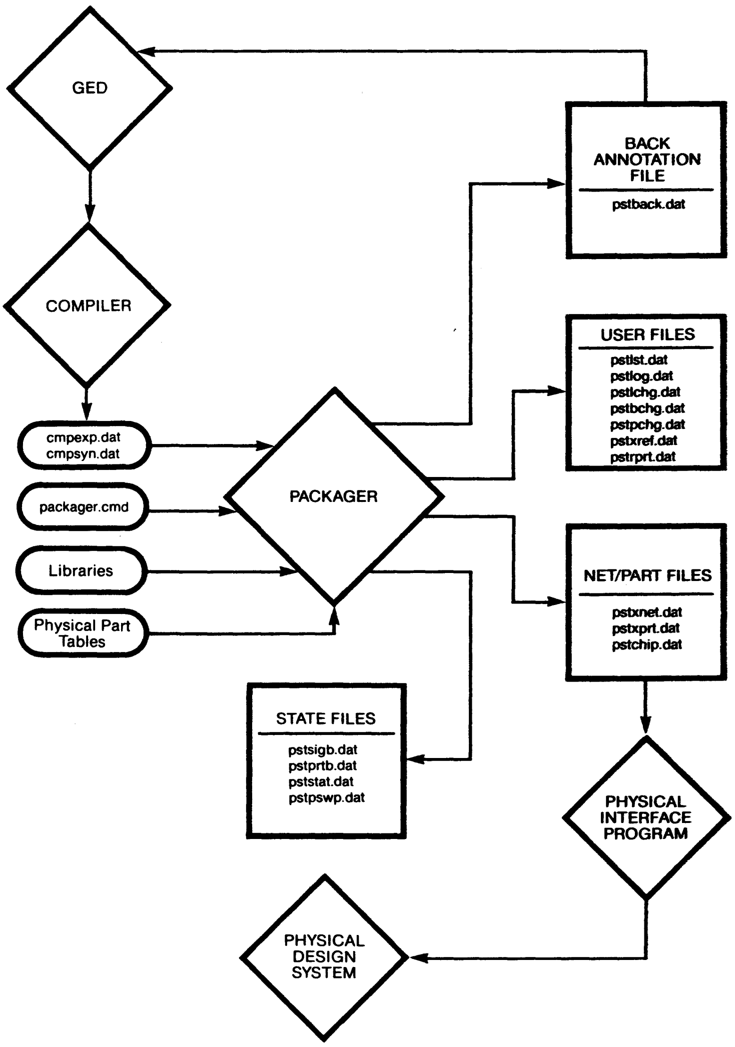

Figure 2-2. First Run of the Packager

Net/Part Files: to be processed by interface programs before going to physical layout.

- Expanded Net List (PSTXNET). Lists each net on your design (alphabetically by signal name) and the nodes connected to it (by U-number).

- Expanded Parts List (PSTXPRT). Lists each physical part in the design in order (by U-number) and tells what logical part (by PATH property) is assigned to each section.

- New CHIPS File (PSTCHIP). Lists the physical information from the library chips files and physical part tables for each different library part that your design utilizes. This information is excerpted from the library chips files. Since this file is much shorter than the library chips files, it is passed on to subsequent programs for them to use.

User Files: documentation and statistics for the Packager run, for user's reference.

- Listing File (PSTLST). Provides process information and error messages for the user.

- Log File (PSTLOG). Provides process information and other data for use by Valid personnel.

- Cross References (PSTXREF). Lists, for cross reference purposes, signal names and the net names to which they correspond, and logical part names (library part names and PATH properties) and the physical assignment (U-number, section, and pin numbers) to which they correspond.

- Logical Changes Summary (PSTLCHG). Lists the logical parts that were added or deleted from the design since the last run of the Packager.

- Physical Changes Summary (PSTPCHG). Lists all physical parts that were added to the design or deleted from the design during this run.

- Binding Changes Summary (PSTBCHG). Lists all bindings that were added to the design or deleted from the design during this run. (A binding is a mapping of a logical part to its allocated physical section.)

- Reports File (PSTRPRT). Lists the remaining spare sections (if any), and how many packages of each physical part your packaged design requires.

Back Annotation File: for back annotation through GED.

- Back Annotation File (PSTBACK). Lists the information in the expanded net list and the expanded parts list ordered by body name and PATH property so that GED can write in physical part designators (e.g., U-numbers) and pin numbers for each body on the drawing.

Output Files for Use in Later Packager Runs (State Files)

- Logical Signal Name to Physical Net Name Binding (PSTSIGB) and Logical to Physical Part Designator Binding (PSTPRTB). These two files contain the information in the expanded net list and the expanded parts list, in a somewhat different format.

- State File (PSTSTAT). This brief file time-stamps the current Packager run and identifies the compilation run used for input.

- Pin Swap File (PSTPSWP). This file lists the pins swapped during the current Packager run. If no pins have been swapped, the file includes only a header and "end."

Section 13 contains detailed descriptions of Packager output files and includes a complete discussion of file format and syntax.

The back annotation file is used by the Graphics Editor (GED) to annotate the schematic with the physical part designators (e.g., U-numbers) and pin numbers. See Section 7 for a detailed discussion of back annotation.

2.2 SUBSEQUENT PACKAGER RUNS

Figure 4 shows the overall data flow for subsequent Packager runs. Notice how the state files from the previous Packager run become input for the next run. The state files ensure consistency in the Packager's physical assignments from one run to the next. Section 3 discusses state files. The feedback files can come from the physical design system or can be created manually. Feedback files typically contain changes in U-numbers and pin assignments to optimize the design and shorten wire lengths. You can also change net names and swap sections. The Packager then incorporates these changes. See Section 6 for a detailed discussion of feedback files.

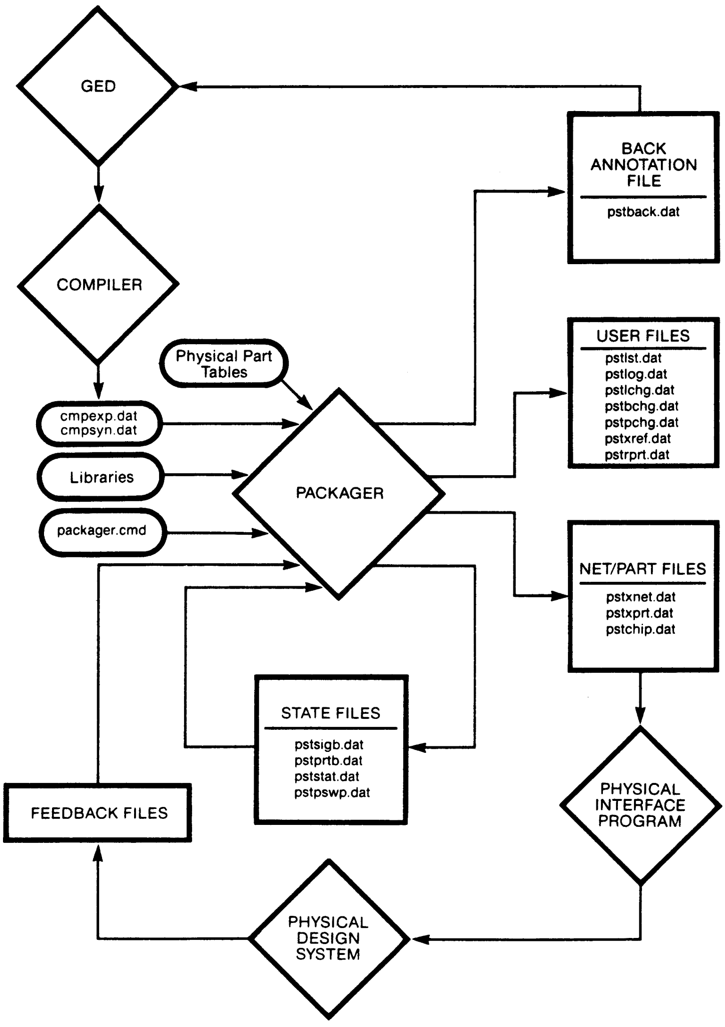

Figure 2-3. Subsequent Packager Runs

2.3 THE PACKAGER LISTING FILE

An example of a Packager listing file is shown in Figures 5A and 5B. The listing file is contained in PSTLST. The listing consists of several parts. The paragraph numbers below correspond to the "call out" numbers on Figure 5.

1. The first part of the listing file is the header. It tells you which version of the Packager you used, and the time and date of the Packager run.

2. The next part of the listing file is the directives list. It tells you every directive that was in effect for this run. This includes all the directives you entered in PACKAGER.CMD as well as the default settings for the directives you omitted.

3. The next part of the listing gives the drawing name and the date and time it was compiled. This information is taken from the Compiler expansion file.

4. The next several items on the listing file are process statements. As the Packager does its work, it reports at each stage. Errors are noted as they are found.

5. The listing file ends with a recap of the number of errors, oversights, and warnings found by the Packager, and the elapsed time and CPU time for this run.

```
     Valid Logic Systems, Inc.  Packager  7.25h:1June85

  1  Packager run on Fri Jun  7 16:43:51 1985   at 16:43:51.00

     ********************************
     *  Starting to read directives  *
     ********************************

     ------ Directives ------

     WARNINGS ON;
     USE_PIN_GROUP ON;
     SUPPRESS <none>;
     OVERSIGHTS ON;
     DOCUMENT_ERRORS ON;
     OUTPUT EXPANDEDNETLIST,
            EXPANDEDPARTLIST,
            LOGICALCHANGES,
            LOCALPARTXREF,
            GLOBALSIGNALXREF,
            GLOBALPARTXREF,
            BACKANNOTATION;
  2  LIBRARY_FILE '/u0/lib/tutorial/tutorial.prt';
     INCLUDE_IO_LIST OFF;
     REPORT SPARES,
            PARTSUMMARY;
     ANNOTATE BODY,
              PIN;
     USE_STATE_FILES ON;
     PART_NAME_LENGTH 16;
     NET_NAME_LENGTH 24;
     UNNAMED_CHANGES ON;


     *************************************
     *  Starting to read compiler output  *
     *************************************

     ------ Expansion file information ------

  3  ROOT_DRAWING='SUBTRACTOR';
     TIME=' COMPILATION ON FRI JUN  7 16:38:55 1985  ';


     ******************************************
     *  Starting to read library description  *
     ******************************************

     ********************************
     *  Starting to read state file  *
     ********************************
  4
     **********************************************
     *  Starting assignment of SIZE replicated parts  *
     **********************************************

     ****************************
     *  Starting TIMES replication  *
     ****************************

     **************************
```

Figure 2-4A. Packager Listing File

```
*  Starting to thread nets  *
*****************************

****************************************
*  Starting to assign physical parts  *
****************************************

********************************
*  Starting to evaluate nets  *
********************************

*  Starting to check nodes  *
*****************************

*********************************************
*  Starting to assign physical part names  *
*********************************************

********************************************
*  Starting to assign physical net names  *
********************************************

**********************************************
*  Starting to assign physical group names  *
**********************************************

************************************
*  Starting to perform pin swaps  *
************************************

**************************************
*  Starting state files generation  *
**************************************

**************************************
*  Starting output list generation  *
**************************************
```

5

```
Packager run on Fri Jun  7 16:43:51 1985    at 16:43:51.00

Design name:
     SUBTRACTOR

Design compilation:
      COMPILATION ON FRI JUN  7 16:38:55 1985

Library creation:
      COMPILATION ON THU JUN  6 11:07:06 1985


No errors detected
9 oversights detected
4 warnings detected

Start time   = 16:43:51.00
Ending time  = 16:44:38.00
Elapsed time = 00:00:47.00
CPU time     = 00:00:42.20
```

Figure 2-4B. Packager Listing File

2.4 ERROR MESSAGES IN LISTINGS

Packager error messages are also included in the listing file. Errors, oversights, and warnings are all listed together. An error is a serious problem which must be fixed before you can continue. An oversight is a less severe problem which should be fixed, but which does not halt progress. A warning is a minor problem which you may or may not choose to fix. Here is an example of a warning message:

```
#2 WARNING(132): No input on net
```

In this example, "#2 WARNING" means that this is the second warning message from the Packager. Errors, oversights, and warnings are numbered separately. (For example, the third error would appear as "#3 ERROR".) The number "132" is the number of the warning message. The Packager has 229 different messages. The text "No input on net" is the warning message itself. See Section 15 for more detailed information on error messages. It contains all messages in numerical order with a descriptive paragraph for each. These descriptions are also printed in the listing file as short paragraphs after the list of errors, one paragraph for each type of error, oversight, or warning. (To omit these paragraphs, use the directive DOCUMENT_ERRORS OFF.)

2.5 COMMONLY-USED DIRECTIVES

The following directives are used frequently:

- FEEDBACK_ORDER - specifies the feedback files and the order in which the Packager will process them. This directive causes feedback to occur.

- LIBRARY_FILE - specifies the libraries containing physical information for parts in the design. This directive or the LIBRARY directive is always required.

- LIBRARY - same function as LIBRARY_FILE, but uses the short library name instead of the full path name.

- OUTPUT - this suppresses and enables selected output files.
- OVERSIGHTS - if set to OFF, this suppresses all oversight messages in the listing.
- PART_TABLE_FILE - specifies the files containing physical part tables to be referenced by the Packager.
- SUPPRESS - this suppresses one or more warning or oversight messages.
- USE_STATE_FILES - controls the generation of state files.
- WARNINGS - if set to OFF, this suppresses all warning messages in the listing.

See Section 12 for detailed descriptions of the format and functions of all Packager directives.

Most Packager directives have default values. Here is the current set of default directives:

```
USE_PIN_GROUP ON;
WARNINGS ON;
SUPPRESS <none>;
OVERSIGHTS ON;
DOCUMENT_ERRORS ON;
OUTPUT EXPANDEDNETLIST,
     EXPANDEDPARTLIST,
     LOGICALCHANGES,
     PHYSICALCHANGES,
     BINDINGCHANGES,
     LOCALPARTXREF,
     GLOBALSIGNALXREF,
     GLOBALPARTXREF,
     BACKANNOTATION;
INCLUDE_IO_LIST OFF;
REPORT SPARES,
     PARTSUMMARY;
```

```
ANNOTATE BODY,
      PIN;
USE_STATE_FILES ON;
PART_NAME_LENGTH 16;
NET_NAME_LENGTH 24;
UNNAMED_CHANGES ON;
```

2.6 INTERFACES TO PHYSICAL DESIGN SYSTEMS

The following output from the Packager is sent to a physical design system:

- Expanded Net List (PSTXNET) - lists each net and its attached nodes
- Expanded Part List (PSTXPRT) - lists each physical part and shows the logical part assigned to each section.
- CHIPS File (PSTCHIP) - physical characteristics of the parts in the design, extracted from the libraries and physical part tables.

These files must be reformatted by a physical interface program before they can be processed by a physical design system. Likewise, the output from the physical design system must be reformatted for feedback to the Packager. Section 6 discusses the feedback process and covers the format of feedback files.

2.7 THE SUBDIRECTORY OPTION

The "package" command which runs the Packager has a subdirectory option which allows you to package different designs in the same directory and keep separate sets of output files for each design. The subdirectory option can also be used to save the current state of the design by keeping a copy of the Compiler expansion file and a copy of the current Packager output files in a subdirectory. In this way, the subdirectory can provide a backup for a Packager run. The format of the "package" command with the subdirectory option is as follows:

package [-s] [*root drawing name*]

The option "-s" tells the Packager to save a copy of the output files from this run in a subdirectory. The name of the subdirectory is pack*i*, where *i* is an integer between 1 and 9. When you run the Packager with the "-s" option, it searches through all subdirectories for a Compiler Expansion file with the same *root drawing name* specified in the "package" command. If none is found, a new subdirectory is created. If the *root drawing name* was not specified, the Packager uses the name in the Compiler Expansion file in the directory. Here are some examples:

If you use the command "package -s" without the *root drawing name*, the Packager saves a copy of the Compiler Expansion file and all Packager output files in a subdirectory. The Packager first searches for an existing subdirectory containing a root drawing name which matches that in the Compiler Expansion file in the current directory. If there is none, then the Packager creates one to store the files.

If you use the command "package -s DESIGN1", the Packager searches for an existing subdirectory with a Compiler Expansion file that contains the root drawing DESIGN1. Note that if the root drawing name contains blanks, you must enclose it in double quotes.

The subdirectory "snapshot" of the design is overwritten every time the design is repackaged with the "-s" option. If you want to save several different copies of the same design, you must rename the drawing in GED so that the root drawing name is different from the previous Packager run.

SECTION 3
STATE FILES

A small change in the logical design should cause only a small change in the physical design. This makes it possible to modify a design while physical design is in progress without redoing the physical layout. State files are used to keep Packager assignments consistent from one run to the next.

State files provide the Packager with the assignments from the previous run. Those assignments which are still legal in the current run are performed. Any new logical parts, nets, or pins are then assigned. (An assignment is legal if the parts, nets, or pins it references still exist in the logical design.)

The Packager generates and reads the part bindings (PSTPRTB), signal bindings (PSTSIGB), pin swap (PSTPSWP), and design information (PSTSTAT) state files. These files record the logical to physical part allocation, logical to physical net name assignment, logical to physical pin assignment, the pin swaps, and global design information for the last run of the Packager.

If the use of states files is enabled and these files exist when the Packager is run, they are used to guide logical to physical part allocation and logical to physical net name assignment, and logical to physical pin assignment. The Packager reports whether the state files are being used in the listing file (PSTLST) under the sections "Assign Physical Parts", "Assign Physical Net Names", and "Perform Pin Swaps".

3.1 USING STATE FILES

When state files are enabled, the Packager reads them if they exist, and generates them after the logical to physical assignments have been completed. Since the state files for any design are named PSTPRTB, PSTSIGB, PSTPSWP, and PSTSTAT, it is necessary to keep each design in a separate directory, or for IBM mainframe systems, a separate disk. This ensures that a state file for one design will not be applied to a different design.

If you want to disable the use and generation of state files, you must use the Packager directive USE_STATE_FILES OFF, since the default is ON. In general, you should use state files to preserve the consistency of the design from one run to the next.

Since the assignments specified in the state files are based on the history of the design, the component packing which they specify may not be as tight or as regular as that the Packager might generate from scratch. Deleting the odd bits of a bus might, for example, result in a set of buffers where every other section is used. If common pin usage allows, new logical parts will be allocated to the unused sections, but this may result in a physical design which is difficult to wire.

For these reasons, you may choose to delete the state files occasionally, to allow the Packager to repack the design in the most compact form. Deleting the state files WILL GREATLY CHANGE THE ASSIGNMENTS for a design, and may result in slightly different loading on nets connecting to common pins. Obviously, the state files should NEVER be deleted for designs which have already been built. Rather than deleting the state files, they should be SAVED so that they may be used if the new assignments are for some reason undesirable.

The state files should never be edited in order to change the physical design. Only the feedback files should be used to alter the design. Refer to Section 6 for more information on feedback.

SECTION 4
OVERVIEW OF PACKAGER FUNCTIONS

This section contains a brief description of the following Packager functions:

- Logical to Physical Assignments
- Creating Net and Parts Lists
- Creating PSTCHIP File for Physical Design Systems
- Expansion of Structured Parts
- Removing Phantom Wire-OR and Wire-AND Bodies
- Load and Net Checks

These functions are covered here to provide a better overall understanding of the Packager. They are discussed in more depth later in the chapter.

4.1 LOGICAL TO PHYSICAL ASSIGNMENTS

The Packager transfers a design from the logical realm to the physical realm. During this transfer, the Packager makes the following assignments:

FROM LOGICAL DESIGN TO PHYSICAL DESIGN		
logical parts	are assigned to	physical sections
logical signals	are assigned to	physical nets
logical nodes	become	physical pins

The Packager assigns the logical parts to physical chips ("packages") which will be placed on a printed circuit board. A logical part is a body on the drawing created through the Graphics Editor. The Packager assigns logical parts to a section of a physical part. Physical parts are named by physical part designators (e.g., U1, U2, U3).

Note that logical part names and physical part designators have no correspondence. That is, there is no connection between the two names.

Physical net names are created from an abbreviation of the logical signal name for the net. (See Section 8 for a detailed description of the rules for abbreviating logical signal names.) If there are several signal names on one net, the Compiler picks one for the Packager to use.

The Packager assigns logical nodes to physical pins on a chip (e.g., A0 becomes pin 5, CE becomes pin 1). The Packager uses the library model of the logical part to determine the pin numbers. See Section 9 for more information on library models and pin number assignments.

4.2 CREATING NET AND PARTS LISTS

The Packager creates expanded net and parts lists which contain all the information about the physical design. These lists are organized by physical information, and they can be sent to a physical design system. (They are usually reformatted by a physical interface program before going to the physical design system.) The user cross reference file (PSTXREF) contains the same information in a more accessible form.

The Expanded Net List is ordered by physical net name and contains the properties for each net and the logical to physical binding of nets and nodes as well as node properties. The Expanded Part List is ordered by physical part name and contains the properties and logical to physical bindings of each part.

Section 13 has sample expanded net and parts listings and detailed explanations of the file format. It also contains a detailed description of the user cross reference file (PSTXREF).

4.3 CREATING THE PSTCHIP FILE

The Packager uses information from libraries and physical part tables to determine the physical characteristics of the various parts in a design. The Packager extracts information from these files for each part in the design and places it in the output file PSTCHIP. This output file can then be used by physical design systems.

Library files contain information attached to the body. This information includes the pin numbers for the part, the input and output loads for each pin, and the family of the part.

Physical part tables provide a way to create new part types from a basic part type. For example, you can create many different types of resistors and capacitors from a single basic resistor or capacitor. The various resistor types may have different resistance values, power dissipation, cost, reliability, etc. All of these characteristics can be specified in a physical part table. There is only one library definition for the part, and therefore only one copy of the models. The Packager uses the properties attached to the part to differentiate it from other instances of the same part.

Physical part tables thus allow you to attach new body properties to a part type without having to recreate or modify the library files containing the part type definitions. An important use of this capability is the addition of new properties to the libraries for certain interfaces to physical design systems. These properties describe to the interface the type and shape of each component.

4.4 EXPANSION OF STRUCTURED PARTS

The Packager expands any structured parts present in the logical design. A logical part with the SIZE property is expanded into the corresponding number of parts, and the vectored signals connected to the ''sized'' part are assigned to the expanded parts. The Packager also creates new versions of parts with TIMES properties.

The SIZE property is used to generate a multiple-bit component and to connect it to a group of signals. The TIMES property is used to replicate an output that must drive many inputs. See Section 8 for more information on SIZE and TIMES.

4.5 REMOVING PHANTOM BODIES

Figure 4-1 shows an example of "phantom" wire-OR and wire-AND bodies. These bodies are simply representations that you may use in your design to document the logical operation that is performed at a wire-tie. Phantom bodies may be accessed with the Graphics Editor by entering the command "library phantom." Of course, these are not "real" gates, and the Packager replaces them with wire-ties (also called "wire-gates"). For more information on this topic, see Section 8.

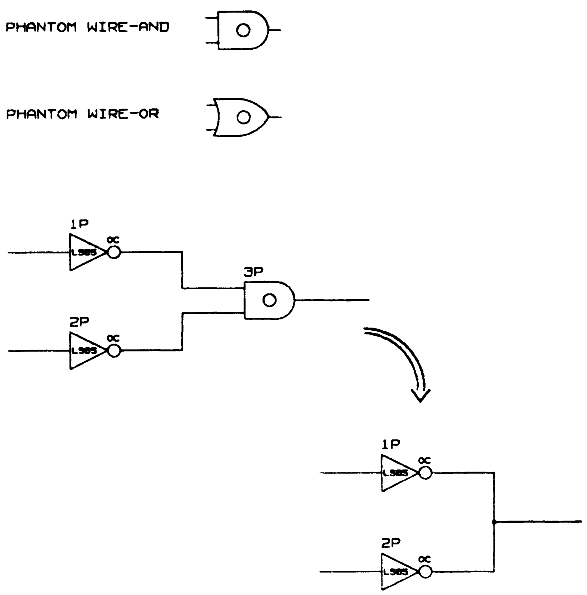

Figure 4-1. Wire-OR and Wire-AND Phantom Bodies

4.6 LOAD AND NET CHECKS

A net is a single bit signal and the nodes that are connected to it. The Packager performs the following net error checks:

- Output-Type Check: make sure that output pins connected together have the proper technology (e.g., open collector, open emitter, tri-state).

- I/O Check: make sure every net is connected to at least one input pin and at least one output pin.

The Packager calculates the loading for each net for both logic states ("0 state" and "1 state"). The Packager performs the following load checks:

- For each logic state, the input and output loads must have opposite signs.

- For each logic state, the absolute value of the smallest output load must be greater than or equal to the absolute value of the total input loads on that net.

See Section 8 for a complete description of the Packager's net and load checks. This section also discusses properties which suppress load checks in certain cases (e.g., unknown loading).

SECTION 5
USER DIRECTIONS FROM THE SCHEMATIC

There are often cases where you need to direct the assignments made by the Packager. For example, you may want to assign four gates performing a particular function to the same package, and there is no guarantee that the Packager will make such an assigment. To solve this type of problem, there are several properties and commands you can use on your schematic which affect the Packager's assignment scheme. These include the following:

- LOCATION Property
- SECTION Command
- GROUP Property
- LOCATION_CLASS Property
- PINSWAP Command
- FLAG Bodies and Interfaces

This section discusses each of the above properties and commands in the order listed above. This is followed by a paragraph containing suggestions for using these properties most effectively. You can use the above properties and commands on your logical design in the Graphics Editor. There are several other properties which are used at the "body" level when new parts are added to a library (e.g., PIN_NUMBER, PIN_GROUP). See Section 9 for more information on these properties and how they affect the Packager. The Graphics Editor reference chapter contains detailed instructions on the use of these commands and properties in GED.

The following properties do not affect the Packager assignment scheme but do affect load and net checks or physical part designators:

- UNKNOWN_LOADING - indicates that device loading is not known

- NO_LOAD_CHECK - suppresses device loading calculations

- NO_IO_CHECK - suppresses input and output net checks

- ALLOW_CONNECT - permits an output pin to be connected to a net regardless of other outputs on the net

- PHYS_DES_PREFIX - specifies the prefix to use for the physical part designator. This can be defined in the library to set the default prefix for that part. This property does not affect the physical part designator if the part has already been named in a previous Packager run and if the current Packager run is using State files. If the part has already been named and you must use State files, then use the LOCATION property to change the physical part designator.

5.1 LOCATION PROPERTY

Use this property to assign a particular physical part name to a logical gate on a design. The LOCATION property always takes precedence over a physical part name assigned by the Packager. If you assign the same name to several gates, this has the effect of grouping them together; however, grouping is NOT the main purpose of this property. Use the GROUP property if you just want to group parts and don't have a particular physical part name to use. Figure 7 demonstrates the use of the LOCATION property. Note that it is an error to assign the same LOCATION name to more gates than will fit in the same physical part.

The LOCATION property can be attached only to physical part bodies. LOCATION properties attached to hierarchical drawings are errors and are ignored. The LOCATION property is not inherited as a body property.

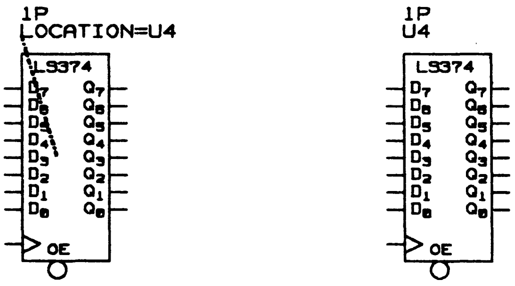

Figure 5-1. Use of the LOCATION Property

5.2 SECTION COMMAND

This command assigns a logical gate to a particular section within a physical part. The SECTION command does not specify the particular physical part; it simply assigns the gate to a particular section within a part of a given type. You can use the LOCATION property and SECTION command together.

Currently, the only parts that can be assigned to a particular section must have a SIZE of 1 or have the property HAS_FIXED_SIZE.

5.3 PINSWAP COMMAND

A swappable group of pins are those pins which are logically equivalent and belong to the same section. This means that if two nets are swapped between two pins that are in a swappable group, the logical function of the circuit is not altered. The PINSWAP command allows you to perform this operation on sectioned bodies.

A common example of this occurs for the inputs of a NAND gate such as a 74LS00. The two input pins are physically equivalent in terms of loading and propagation delay from input to output. Thus, if the nets to the input pins are swapped, the behavior of the circuit is unchanged.

A set of pins on a given part that are swappable must have the same value for the PIN_GROUP property attached to them. Any pin without the PIN_GROUP property is not swappable with other pins. The value of the PIN_GROUP property is not important, but all pins of a swappable group must have the identical value. If you want to swap pins which do not have the PIN_GROUP property, use the directive PIN_GROUP OFF to permit this.

The Packager and the section and pin assignment program used by the Graphics Editor recognize the PIN_GROUP property on pins of parts in the CHIPS files. The property is used to assign the logical pins to pin equivalent and swappable groups so that the Packager can perform legal pin swaps.

5.4 GROUP PROPERTY

This property allows you to group logical parts together into the same set of physical packages. Logical parts with different GROUP properties (or no GROUP property) are placed in different physical packages. For example, in Figure 8 the NAND gates with GROUP = A will all be placed in the same set of physical packages, say U1 and U2, those gates with GROUP = B will be placed in U3, and those gates with no GROUP property will be placed in U4. Grouping allows you to force physical allocation yet not be concerned with specific physical part designators (e.g., U-numbers). The GROUP property applies to the entire design.

When using the GROUP property, you do not need to know the name of the physical package, and you also do not need to keep track of the number of gates in a group. If there are more gates in a group than will fit on a single part, the surplus gates automatically spill over to another part of the same type.

If you want to mix grouped parts with parts which are not grouped, use the directive FREE_GROUPING ON. Otherwise, use the directive FREE_GROUPING OFF (the default is ON). This ensures a tight placement, yet still keeps distinct groups on separate parts. For example, in Figure 5-2 FREE_GROUPING ON would place the NAND gates with no GROUP property into either U2 or U3.

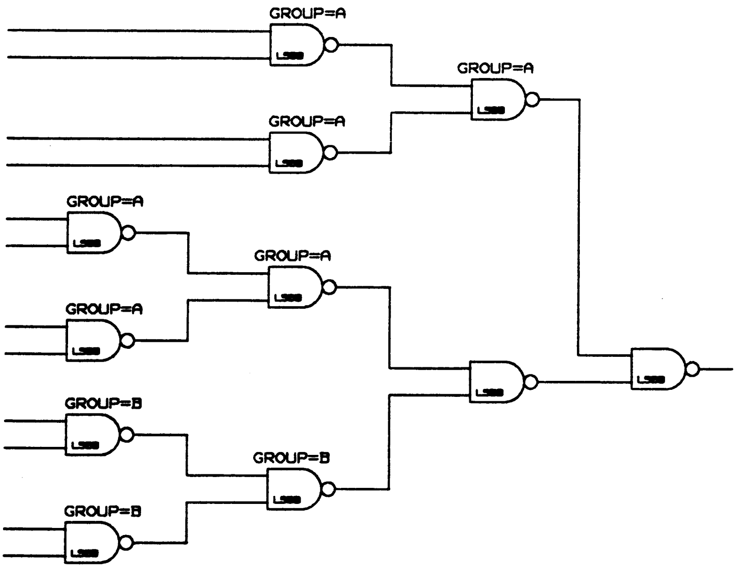

Figure 5-2. Use of the GROUP Property

Note that you can use LOCATION, GROUP, and SECTION on the same logical part, but you should be very careful in your assignments. For example, you cannot assign the same LOCATION to parts in different groups. However, parts in the same group can be in different physical packages. Note that you cannot use both GROUP and LOCATION_CLASS in the same design.

If the State files are used and some members of a group are deleted after their initial assignment, on subsequent runs the Packager will assign the remaining members of the group according to their previous assignments as recorded in the State files.

5.5 LOCATION_CLASS PROPERTY

This property is similar to GROUP. The difference is that logical parts without the LOCATION_CLASS property are always used to fill physical packages that contain parts with a LOCATION_CLASS property. This capability is now replaced by using the GROUP property and the directive FREE_GROUPING ON. Note that LOCATION_CLASS works on a page of the design, whereas GROUP applies across the entire design. You cannot use both GROUP and LOCATION_CLASS in the same design.

5.6 FLAG BODIES AND INTERFACES

Interfaces between a circuit and its external components must be defined in some manner. In board level designs this is normally done by implicitly defining interface signals through connectors spread throughout the design. Many gate array design systems, however, treat interface signals differently from internal signals. Often they must be declared in separate parts of the net list and must have extra information attached. For this reason, the Packager must be able to distinguish interface signals and treat them accordingly.

The Packager supports the use of FLAG bodies in the drawings to define the connection of interface signals in a root drawing to some external component such as a gate array chip carrier. To define a signal as an interface signal, attach a FLAG body to the signal. When the INCLUDE_IO_LIST ON directive is specified, the Packager will attach the IO_NET property to the interface signals with the value INPUT, OUTPUT, or BIDIRECTIONAL, as defined by the FLAG body. These properties are then output in the Expanded Net List.

5.7 SUGGESTIONS FOR USE

Here are some suggestions for effective ways to use these commands and properties:

1. Before the first Packager run, use the GROUP property to group related parts. If you are certain of the names for particular parts, use the LOCATION property. Likewise, if you are sure of the placement of certain sections within parts, use the SECTION command. Otherwise, don't use LOCATION and SECTION.

2. If you want to change your LOCATION or SECTION assignments during a feedback run, use the directive HARD_LOC_SEC OFF. Then be sure to back annotate and recompile the design to keep everything synchronized. Likewise, to move a section from one group's physical part to another group's physical part during feedback, use the directive HARD_GROUP OFF and back annotate. Be sure to recompile before repackaging the design. Be very careful when you use these directives because they leave all LOCATION, SECTION, or GROUP assignments open to change. Inadvertent feedback errors can be very damaging in these circumstances. Also, note that if you change a location or section assignment in this way, the LOCATION and SEC properties are removed from the affected logical parts, while GROUP properties are changed according to the new assignments.

SECTION 6
FEEDBACK TO THE PACKAGER

The physical design generated by the Packager may not be optimal for layout. The physical design system may rearrange parts, swap equivalent sections within parts, and swap equivalent pins on a section. Without feedback and back annotation, these changes will not appear on the schematic. Also, if there are further modifications to the logical design, the Packager and the physical design system will be "out of synch."

Feedback files inform the Packager of physical design changes. In this case, the files are usually generated automatically by the physical design system. Sometimes you may want to make manual changes to the Packager's physical design through feedback files. For example, you may want to make a small change without going through GED and recompiling. If you do this, remember that you must keep the Packager listings and physical design system files synchronized. It is also a good idea to back annotate the physical changes.

This section discusses the following topics:

Types of Feedback
Warnings
Running the Packager with Feedback
Feedback File Formats

6.1 TYPES OF FEEDBACK

There are four types of changes that can be made through feedback files:

1. Physical part name changes

   When a design is laid out, the physical part designators are often changed to include position information. A typical scheme is to give a part a name of the form *letter number* where *letter* and *number* represent coordinates in two dimensions on a board. For example, G13 could represent row G, column 13.

2. Physical section reallocation

   To simplify wiring, it is often desirable to group together those parts that connect to each other. Since the sections in a part may connect to different groups of parts, it is sometimes necessary to move a section from one part to another part of the same type. If all sections of the destination part are in use, then it is necessary to move one of them somewhere else. This process is often done by swapping two sections between different parts of the same type. Sections are sometimes reassigned within a single part to improve wiring.

3. Physical pin reallocation within a section

   To simplify wiring even more, equivalent input pins of a section may be reassigned or swapped. This is frequently done to parts having many equivalent inputs.

4. Physical net name changes

   Changing physical net names does not affect the layout or wiring of a design, but users may wish to rename nets for documentation or standardization reasons.

Note that you cannot change part types through feedback. You must use the Graphics Editor (GED) to change the part type on the logical design and then recompile and repackage. For example, you cannot swap a 74LS00 for a 54LS00 or 74LS32 through feedback; this change must be made on the logical design.

If the directive HARD_LOC_SEC is set to OFF, then you can change section and location assignments through feedback files even if the assignments were attached as properties in the Graphics Editor. Also, if HARD_GROUPING is OFF, you can reassign sections without regard to their groups. However, be sure to back annotate to keep the logical design in GED synchronized with the Packager's physical design. You must also recompile the design before running the Packager again.

The Packager currently can process four types of feedback files. You have the freedom to use any or all of the files as the situation requires. Only these files should be used to change the physical design. You should never edit the state files. The feedback files are as follows:

PSTPRTX - Physical part designator transformations file

> Use this file to rename a physical part. The file contains a list of old physical part designator and new physical part designator pairs.

PSTSECX - Physical section transformations file

> Use this file to reassign a logical part from an old physical section to a new physical section. The file contains a list of old-physical-section to new-physical-section pairs.

PSTNETX - Physical net name transformations file

> Use this file to change physical net names. The file contains a list of old physical net name and new physical net name pairs.

PSTFNET - Feedback net list

> Use this file for physical part designator changes, physical section swapping, and pin swapping. You cannot change physical net names with this file. The Feedback Net List file is frequently used by physical design systems to make several types of changes at one

time. If you are generating feedback files manually, use the Feedback Net List file only if you are swapping pins. Otherwise, use the other feedback files, since they have simpler formats and are easier to use.

6.2 COMMON ERRORS TO AVOID

1. Never recompile a design between the time it was packaged for a physical design system and the time you use feedback from that system. The Packager uses the Compiler expansion and synonym files, the Library files, and State files during feedback processing. If any of these have changed since the previous Packager run, this will generate errors. You should save the following files for any design that is sent to a physical design system:

   Compiler output files
   Library files
   Physical part tables
   Packager state files

2. Never add or delete a part in the physical design system. Also, never change a net name. Many physical design systems are capable of making these changes, but the function of the design must be controlled from the Graphics Editor.

3. The directive NET_NAME_LENGTH should not be changed from the previous Packager run to the feedback run. This ensures that net names remain the same.

6.3 RUNNING THE PACKAGER WITH FEEDBACK

Feedback requires the state files from the previous Packager run. Be sure to include the directive USE STATE_FILES ON in the directives file when you plan to do feedback on the next Packager run. This directive is also required for the feedback run.

The FEEDBACK_ORDER directive specifies the type of feedback files to be used and the order in which the Packager will process them. When you include this directive in the directives file, you are telling the Packager to perform feedback. The format of this directive is as follows:

feedback_order [-v] *filetype* [,*filetype*] ... ;

where *filetype* can be:

part_trans	Physical part designator transformations file (pstprtx) - to change part names
section_trans	Physical section reallocation file (pstsecx) - to change part names and section connections
feedback_netlist	Feedback net list (pstfnet) - to change pin connections, section connections, and part names
net_trans	Physical net name transformations file (pstnetx) - to change net names

For example,

FEEDBACK_ORDER NET_TRANS, FEEDBACK_NETLIST;

specifies that physical net name feedback occurs first, followed by feedback net list changes.

The ''-v'' (verbose) option generates additional error information for certain errors found during feedback processing. For example, error #149 ''Match not found for feedback section'' generates a list of the closest partial matches for the entry. Normally, only the first three partial matches are listed. The "-v" option outputs all partial matches in the listing file (pstlst).

You should be careful to list the feedback files in the correct order. For example, if you refer to new net names in the Feedback Net List file, you should list NET_TRANS before FEEDBACK_NETLIST. In general, do part name changes first, section reallocations second, and pin connection changes third.

6.4 FEEDBACK FILE FORMATS

This section describes the formats of the four feedback files. These are text files that begin with a header and terminate with the marker "END.". The header lines identify the file and the name of the design. The form of the header lines is

FILE_TYPE = *file type* ;
ROOT_DRAWING = '*drawing name*' ;

where *file type* specifies the file's type and *'drawing name'* (enclosed in single quotes) is the name of the root drawing of the expansion file. For example, a header for the Physical Part Designator Transformations file for the drawing SAMPLE would appear as:

```
FILE_TYPE = PART_TRANS;
ROOT_DRAWING = 'SAMPLE';
```

Comments may be placed in the files if enclosed in braces ({ }). A comment may appear anywhere a space may appear. Comments may cross line boundaries but cannot be nested.

If an item is too long to fit on a line (80 characters), it must be broken into more than one line. A tilde (˜) is used as a continuation character to indicate that the current item is continued on the next line. A line break can appear between any two characters in the file. A tilde, however, is only significant if it occurs at the end of the line.

PHYSICAL PART DESIGNATOR TRANSFORMATIONS (PSTPRTX)

Use this file to rename physical parts. The physical part designator identifies a particular occurrence of a physical part. The file type is PART_TRANS, and it consists of a list of transformations in the form

'*old physical part designator*' '*new physical part designator*'

where *old physical part designator* is the physical part designator assigned by the Packager during its last run and *new physical part designator* is the new physical part designator to be assigned. For instance, the physical part designator U31 can be changed to U32 as follows:

```
FILE_TYPE = PART_TRANS;
ROOT_DRAWING = 'SAMPLE';
'U31' 'U32'
END.
```

PHYSICAL SECTION TRANSFORMATIONS (PSTSECX)

No layout knowledge is used during the initial Packager section assignment. When more reasonable section assignments are known, they can be given to the Packager which will use that information to reassign sections. The Physical Section Transformations file is used to specify section changes; its file type is SECTION_TRANS.

The file contains a list of old physical pin designators (as assigned by the Packager during its last run) and new physical pin designators. A physical pin designator consists of a physical part designator and a UNIQUE pin number of the section (not a common pin). The Packager reassigns the sections as specified in this file.

Logical parts with the LOCATION and/or SEC properties cannot be reassigned unless you use the directive HARD_LOC_SEC OFF. This is also true for GROUPed sections unless you use the directive HARD_GROUPING OFF. If you use these directives to make such changes, you must back annotate the design so that the changes will be reflected on the schematic. Also, you must then recompile the design before running the Packager again. The only other way to change section assignments assigned through a property in the logical design is to change the logical design through GED.

The file consists of a list of transformations in the form

'old part name' old pin number 'new part name' new pin number

where *old part name* and *old pin number* specify the current section assignment, and *new part name* and *new pin number* specify the new section assignment. For example, given a 74LS00 (quad NAND gate), a swap of the first two sections on the part U31 might appear as follows:

```
FILE_TYPE = SECTION_TRANS;
ROOT_DRAWING = 'SAMPLE';
'U31' 1 'U31' 4
'U31' 4 'U31' 1
END.
```

FEEDBACK NET LIST (PSTFNET)

You can use this feedback file to swap pins, reassign sections, and rename physical parts. Ycu CANNOT change net names or physical part types with this feedback file. If you are generating feedback files manually, use the Feedback Net List file only if you are swapping pins. Otherwise, use the other feedback files since they have simpler formats and are easier to use.

The file type is FEEDBACK_NETLIST, and it consists of a list of nodes in the form

> *'physical part designator' 'physical part type' pin number 'physical net name'* ;

where *physical part designator* is the new physical part designator, *physical part type* is the part type of the physical part, *pin number* is the new pin number of the node, and *physical net name* is the name assigned by the Packager to the net where the node is connected.

The file MUST be sorted by physical part designator so that all the pins of a physical part appear together. The ordering of the pins on the part does not matter. An example of this file might appear as follows:

```
FILE_TYPE = FEEDBACK_NETLIST;
ROOT_DRAWING = 'SAMPLE';
'U1' 'LS08' 1  'A0';
'U1' 'LS08' 2  'B0';
'U1' 'LS08' 3  'C0';
'U1' 'LS08' 4  'A1';
'U1' 'LS08' 5  'B1';
'U1' 'LS08' 6  'C1';
'U1' 'LS08' 8  'C2';
'U1' 'LS08' 9  'A2';
'U1' 'LS08' 10 'B2';
'U1' 'LS08' 11 'C3';
'U1' 'LS08' 12 'A3';
'U1' 'LS08' 13 'B3';
END.
```

NOTE: If you rename a part, you must enter all the pins for that part since omitted pins are assumed to be 'NC' (not connected). Also, if you reassign a section, you must list every pin in that section.

PHYSICAL NET NAME TRANSFORMATIONS (PSTNETX)

This file is used to change the name of a physical net, and it has the file type NET_TRANS. Each net is originally assigned a name by the Packager that can be changed with this file. The file consists of a list of transformations in the form

'*old physical net name*' '*new physical net name*'

where *old physical net name* is the name assigned to the net by the Packager in the last run and *new physical net name* is the new name to be assigned to the net. For example, the net N00001 can be changed to XYZ as follows:

```
FILE_TYPE = NET_TRANS;
ROOT_DRAWING = 'SAMPLE';
'N00001' 'XYZ'
END.
```

SECTION 7
BACK ANNOTATION

Information usually flows from the drawings of the logical design, through the Compiler and Packager, and on to a physical design system. However, there is an important class of information that flows from the end of this process to the beginning. The Packager and the physical design system add physical data to the design that you may wish to see reflected in the drawings, such as the physical part designator for each part and the pin number for each pin. The process of taking information created or added downstream in the design process and bringing it upstream is called "back annotation."

7.1 WHEN TO BACK ANNOTATE

There are two different times in the design cycle when it is important to back annotate your GED drawing. The first of these is after the first error-free run of the Packager, and the second is after the design has been sent to a physical design system (producing feedback files) and the design has been repackaged to reflect the changes in physical assignments. The following shows the typical order of design steps from the Packager through a physical design system including back annotation.

```
GED
COMPILE
PACKAGE
BACKANNOTATE
(physical interface program)
(physical design system)
PACKAGE
BACKANNOTATE
```

The first time you back annotate your design is after your first error-free run of the Packager. After you have corrected any errors found by the Packager in your design, and run the Packager on the corrected design, you want to record the physical assignments (the U-numbers and pin numbers) on your GED drawing for reference. Remember, however, that these physical assignments may be modified by your physical design system. That is why you will have to back annotate again later.

After this first back annotation, you send the Packager output to a physical interface program to format it for a physical design system, and then to a physical design system. The physical design system creates feedback files that are used as input files to the Packager. You may also create your own feedback files to force the Packager to make certain assignments. You then run the Packager again and it reassigns parts on the basis of the instructions in the feedback files. Now you want to update your GED drawing again so that it corresponds exactly with the physical design you have produced. This is the second time that you back annotate your design.

7.2 HOW TO BACK ANNOTATE

Back annotation brings physical design data from the Packager and adds it to the logical design drawings. The Packager generates a back annotation file that contains physical information grouped by drawing.

To generate this file, use the directive OUTPUT BACKANNOTATION when running the Packager. Backannotation can occur on three types of elements: bodies, pins, and nets. Use the ANNOTATE directive to select among these elements. If this directive is not specified, the default options are bodies and pins. If the net option is specified, the synonyms file from the compilation must be available. The back annotation information is written to the logical file PSTBACK. This file must be renamed to BACKANN.CMD to be used by the Graphics Editor. You can then use the GED command BACKANNOTATE to automatically add all the physical assignments made by the Packager to your drawing. Back annotation saves a lot of time and tedious work and ensures that the drawing accurately reflects the physical part assignments.

It is recommended that you keep a backup of all drawings before back annotation is performed. This will give you both non-annotated and annotated versions of the drawings that may be useful since it does take time to remove the annotated properties from a set of drawings. Figure 7-1 shows an example of a back annotated drawing.

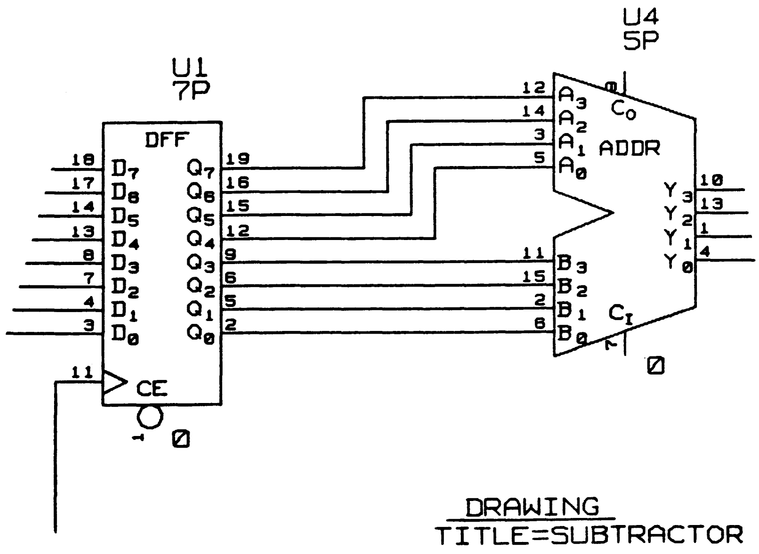

Figure 7-1. Back Annotation of Schematic

7.3 DESIGN TECHNIQUES AND ANNOTATION

If a logical design has structured elements such as SIZE and TIMES, back annotation may not show all the physical assignments on the drawing. Only non-structured elements are annotated. Back annotation cannot be done on structured elements because the physical and logical representations of the drawing are very different. A "flat" drawing has no structured elements and can be completely back annotated. If you have a design that is structured and you must have complete back annotation, you must re-enter the finished design as a flat drawing.

Likewise, if you have a hierarchical design, you can only back annotate the "lowest" levels of the design. Remember that the top level of a hierarchy does not have a one-to-one correspondence to the physical design. For both hierarchical and structured designs, cross reference listings from the Packager (file PSTXREF) may be used to supply the complete set of physical information. These listings are organized such that all data can easily be tracked back to the design drawings. Cross reference listings and the design drawings may be used together for design troubleshooting when a fully backannotated print is not necessary.

7.4 SOFT AND HARD PROPERTIES

You may sometimes notice that certain parts may be updated with new physical information during successive back annotation runs. The properties which may change from one back annotation to the next are called "soft" properties. These are properties which are attached by the Packager program, and they may change from one run to the next.

Properties which you assign are called "hard" properties, and these are not altered by the Packager unless you use the directives HARD_LOC_SEC OFF and/or HARD_GROUPING OFF and modify them through the feedback files. Thus, these properties do not change unless you specifically alter them through feedback or through the Graphics Editor.

For example, you may wish to attach the LOCATION property to a body to ensure that the part is assigned to a specific physical part designator. You would not want this to be changed by the Packager. Thus, LOCATION is a hard property. Other hard properties that you may use are:

GROUP
LOCATION_CLASS
SEC (you assign this with the SECTION command)
PN (you assign this with the PINSWAP command)

Soft properties, on the other hand, are added during back annotation, and they are subject to change on subsequent back annotation runs. You want these to be updated. The Graphics Editor differentiates between hard and soft properties so that it can update information correctly. Soft properties that may be added by the Packager include:

$LOC
$SEC
$PN

Note that soft properties are ALWAYS preceded with a "$" in the property name. Also, soft properties do not force assignments in the Packager; only hard properties can control assignments.

7.5 SPECIAL USES OF PROPERTIES

Although users can assign soft properties, this is generally done by GED during back annotation. However, you may wish to attach a soft property to a .BODY drawing as a placeholder for back annotation purposes. The "$" defines the property as "soft", and the "?" is a placeholder that is later substituted with a U-number.

SECTION 8
PACKAGER FUNCTIONS IN DETAIL

This section contains detailed discussions of various Packager functions which were discussed more briefly earlier in this manual. This section covers the following topics in the order listed below:

Net and Part Name Assignment
SIZE Expansion
TIMES Expansion
Wire-Gate and Wire-Tie Expansion
Net Checks
Load Checks

8.1 NET AND PART NAME ASSIGNMENT

The Packager assigns physical names to both signals and parts. Physical net names are created from the abbreviation of the logical signal name for the net. If there are several logical signal names on a net, the Compiler picks one for the Packager to use. The maximum length of net names is controlled by the NET_NAME_LENGTH directive. The path name portion of the logical signal name is not used in the abbreviation. The abbreviation is created as follows:

1. Remove all special characters. These are all characters except A-Z and 0-9.

2. If the signal is low asserted, add a trailing 'L'.

3. If the name starts with a digit, change it to a letter.

4. If the signal is vectored, append the offset as a number.

5. If the signal is versioned, append 'V' and the version number when the version number is not zero.

6. If the resulting name is greater than maximum net name length, remove all the vowels.

7. If the resulting name is still too long, then truncate the name to the maximum net name length.

8. If the resulting name is not unique, make it unique by incrementing the last non-numeric character of the name. This is to preserve the bit offset that was appended to the name.

For example, the logical net name "READ ADR B<2>" would become the physical net name "READADRB2". Physical part names are created by starting with the value of the PHYS_DES_PREFIX property found on the part type in the library or on the logical part in the design. If there is no prefix on the library part, the Packager uses the standard prefix 'U'. If the name is not unique, it is made unique by adding a number (e.g., U12, U14). The maximum length of the names is controlled by the PART_NAME_LENGTH directive. Note that the PHYS_DES_PREFIX property does not override the old part names from the last Packager run if the current run uses state files. That is, the Packager uses the prefix only if the part is new or if there are no state files.

8.2 SIZE EXPANSION

The SIZE property is used to generate multiple components in one body representation and connect them to a group of signals (a bus). SIZE is used in the logical design to provide a concise means of representing many logical parts as one part together with bus notation. Figure 10 shows an example of a SIZEd part. The Packager generates SIZE number of expanded parts and assigns a new logical designator to each. A logical part can have pins common to all sections as well as pins unique for each section. Common pins are connected in parallel for all sections and the unique pins are connected to independent pieces of the signal connected to the original part. The PIN_NUMBER property for each pin in the body definition specifies whether the pin is common or unique for each section. The PIN_NUMBER property also specifies the width of the pin to which it is attached. The Packager allocates this number of bits from the original signal to each expanded part. The assignment of bits of a signal to physical parts is done sequentially so that adjacent bits are assigned to the same physical package if possible. Figure 8-1 shows an example of the packing of a SIZEd part. See the Library reference manual for more detail.

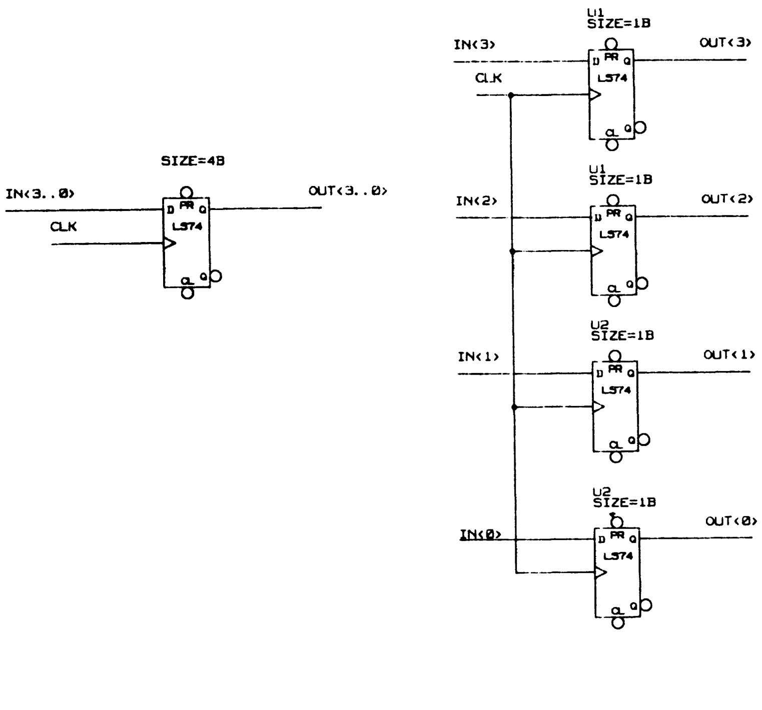

Figure 8-1. SIZE Expansion and Packaging

8.3 TIMES EXPANSION

In digital designs, it is often convenient to have several different signals available which each have the same behavior. One such case is where a net has more input loads than the output is capable of driving. This fan-out error must be corrected if the circuit is to function as designed. A good way to fix this type of problem is to divide the inputs on the net into two or more groups, where each group presents a small enough load to be driven by one output. Each group is wired together and is said to connect to a "version" of the net. To keep the operation of the design unchanged, each version must have the same logical behavior. To avoid fan-out errors, each version of the net must have a different output driving it. This is accomplished by connecting each version of the net to a different version of the output, where each output version behaves the same. Figure 8-2 shows an example.

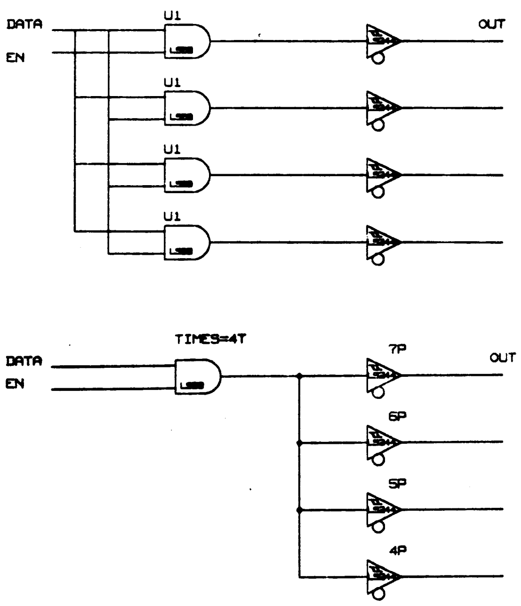

Figure 8-2. TIMES Expansion and Packaging

You can generate multiple versions of the outputs of any part by attaching the TIMES property to the part. The value of the TIMES property equals the number of versions to be generated. The outputs are generated by creating TIMES number of physical sections and, for all physical sections, connecting the inputs as shown in the drawings. Thus, since the inputs to each of the components are identical, each output signal will exhibit the same logical behavior over all the versions.

Replication by TIMES is useful where an output must drive many input loads. The Packager will divide the loads among the versions of an output so that the specified loading rules will be obeyed. In the process, the Packager generates one version of the net for each version of the output. When more versions of an output exist than are necessary to drive the net to which it is connected, the Packager will attempt to divide the loads evenly among all output versions. If loading rules require more output versions than are specified, the Packager determines the number of additional versions which are needed and flags the error.

When several outputs are connected to one net (wire-tied), and several versions of the net are desired, each output should have a TIMES value equal to the number of versions of the net desired. If several outputs on the same net have different TIMES values, the number of versions of the net generated is the minimum of the output TIMES values. Physical parts with no TIMES properties have a TIMES value of one. Figure 8-2 illustrates these rules.

8.4 WIRE-GATE AND WIRE-TIE EXPANSION

For designs that contain parts with connected outputs, you have the option of tying the output signals together in a "wire-tie" or using phantom bodies to explicitly document the logical operation that is taking place. A phantom body is simply a logical representation of a wire-AND or wire-OR gate that is available for use in your design. An example of wire-ties and phantom bodies is shown in Figure 8-3.

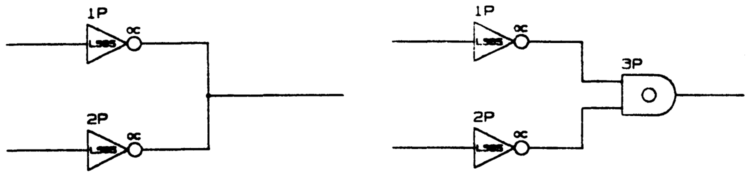

Figure 8-3. Wire-Ties and Phantom Bodies

A wire-tie (or wire-gate, as it is also called) is simply the connection of two or more outputs to the same signal. This is often used to connect several drivers to a common bus. Designers also term a wire-tie a "wire-gate" because a logical operation is performed. A phantom body is used in the design to explicitly demonstrate the function of the wire-tie. When using the TIMES property, phantom bodies also provide a means for using the input signals of the phantom body elsewhere in the design. In this case, a wire-tie will not work because it would connect all signals to the same net. Figure 8-4 illustrates the difference between using wire-ties and phantom bodies with the TIMES property.

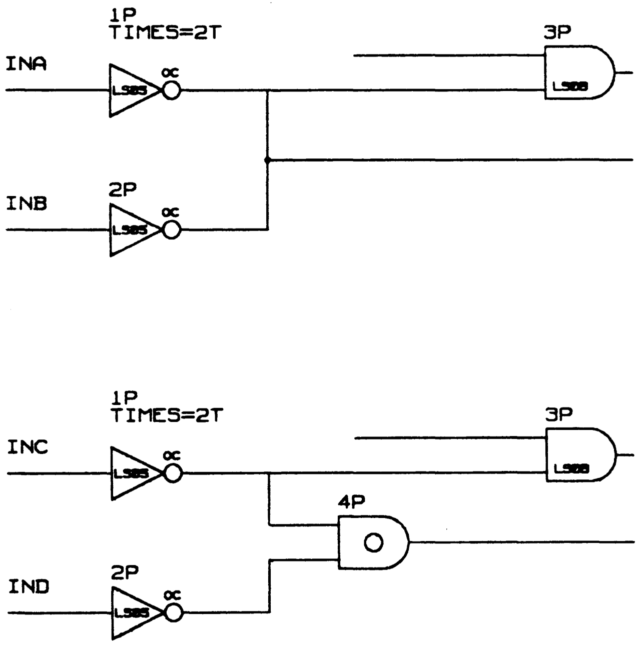

Figure 8-4. Wire-Ties/Phantom Bodies TIMES Property

The phantom body connects versions of the signals used only in the gating function. This way a signal is generated which has the required behavior, and the behavior of the other versions of the constituent signals remains unchanged wherever else they are used. This allows the designer to treat wire-gates the same as "real" gates, which makes complex wire-gate designs understandable.

The Packager replaces phantom bodies with wire-ties (or wire-gates) having first ''expanded'' the design such that the logical function is maintained (see Figure 8-5).

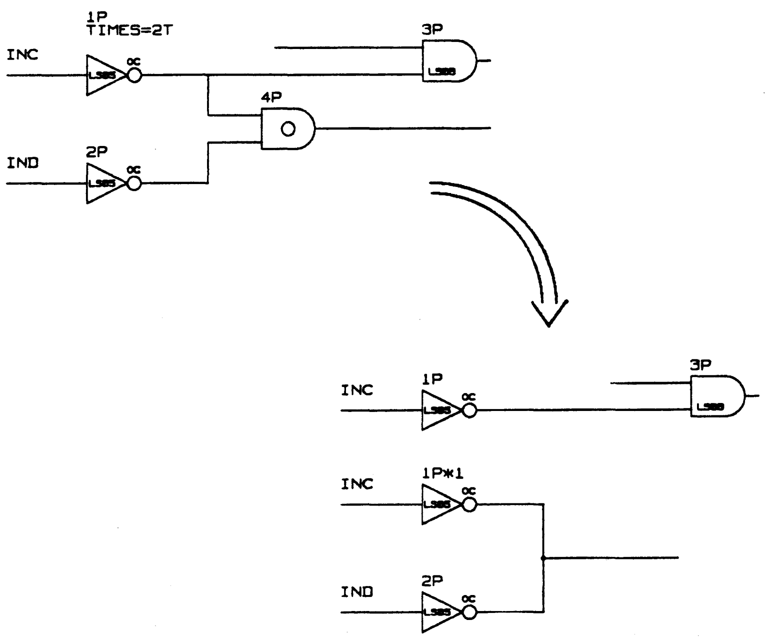

Figure 8-5. Replacing Phantom Bodies with Wire-Ties

TIMES properties on phantom bodies function exactly the same as on physical parts. Several versions of a net are generated, each with different output versions. One version of each of the phantom gate input nets is used for each version of the phantom gate output net. If a net connects to a phantom gate input, then enough versions of the net must exist to drive the ''real'' loads on the net plus the ''induced'' loads on the output net of the phantom gate. If a phantom gate has no TIMES property, it is ignored when determining the number of versions of the output net, and it assumes the TIMES value required by the net. Each version of the output net will use one version of each input net whether the phantom gate has a TIMES property or not. The Packager indicates any extra versions of outputs which are required, making it simple to arrive at the correct TIMES values for outputs driving even a complex combination of "real" and phantom gate inputs.

After a number of versions of a net have been generated by use of the TIMES property, the versions are distributed to wire-gate inputs first. The "real" loads on the net are then divided among the remaining versions of the net. Therefore, a net with 10 versions which gives 3 versions to wire-gate inputs will have 7 versions left to drive "real" inputs.

The constant signals "0" and "1" may be applied to the inputs of phantom gates and function as they would on "real" gates. If a wire-AND has an input connected to the "1" net, that input is ignored, since the other inputs will determine the value of the output. If a wire-AND has an input connected to the "0" net, then all loads on the output net are connected to the "0" net, since 0 AND anything is 0. The same is true for wire-OR's, with "0" and "1" reversed.

8.5 NET CHECKS

A net is a single bit signal and the nodes that are connected to it. The Packager performs three net error checks:

1. Output-Type Check: make sure that output pins connected together have the proper technology (e.g., open collector, open emitter, tri-state).

2. I/O Check: make sure every net is connected to at least one input pin and at least one output pin.

3. Loading Check: make sure loading rules are not violated.

The checks listed above can be suppressed for a signal by attaching various properties to the signal. These properties are discussed in detail later on in this section and are also summarized in a table at the end of this section:

- ALLOW_CONNECT - allows outputs of different types to be tied together (e.g., open collector, tri-state)
- NO_IO_CHECK - suppresses input/output checks
- NO_LOAD_CHECK - suppresses load checks; Packager checks surrounding loads
- UNKNOWN_LOADING - indicates that loading is uncertain; Packager stops load checks for unknown areas

OUTPUT TYPE CHECK

Outputs can only be tied together (making wire gates) if they are given explicit permission to do so. Permission is given either by the OUTPUT_TYPE property included in the CHIPS file for that part, or by attaching the ALLOW_CONNECT property to the output pins.

Parts in Valid libraries have the following standard OUTPUT_TYPE property values:

OC,AND	{ open collector; AND logic function }
OE,OR	{ open emitter; OR logic function }
TS,TS	{ tri-state; tri-state logic function }

Other property values can be used. The value is only used to match output pin types and has no other meaning to the Packager.

When the Packager detects outputs tied together that do not have the OUTPUT_TYPE property or outputs tied together that have incompatible OUTPUT_TYPE properties, it produces an error message indicating the output pins as well as the net name.

Occasionally there is a need to connect outputs of different types. The ALLOW_CONNECT property can be used to allow multiple outputs to be connected together by specifying which outputs are to be "ignored" during the output-type check.

The ALLOW_CONNECT property may appear on a library part (as in the case of a standard connector) or in a logical design. If the ALLOW_CONNECT property is used as a net property, it applies to all the output pins on the nets. When used as a body property, it applies to all the output pins of the body. When used as a pin property, ALLOW_CONNECT applies only to the pin to which it is attached.

INPUT AND OUTPUT PIN CHECKS

The Packager understands that every net must be driven and must also drive. Therefore, each net is checked to make sure that it connects to at least one input as well as one output pin. If this is not the case, a message is printed indicating the condition detected and the net for which it was detected. When parts are created, special properties are attached to their pins to indicate whether the pin is an input, an output, or bidirectional. The INPUT_LOAD property on a pin indicates that the pin is an input, the OUTPUT_LOAD property indicates that the pin is an output, and the BIDIRECTIONAL property indicates that the pin is both an input and an output. These input and output checks can be suppressed on a pin-by-pin, body-by-body, or net-by-net basis. The NO_IO_CHECK property is used for this purpose. The NO_IO_CHECK property can be given one of three values as follows:

LOW
This causes the "0 state" I/O check to be suppressed. The "1 state" check is performed.

HIGH
This causes the "1 state" I/O check to be suppressed. The "0 state" check is performed.

BOTH or TRUE
This causes both the ''0 state'' and the ''1 state'' I/O checks to be suppressed.

The NO_IO_CHECK property may appear on a library part (as in the case of a standard connector) or can be attached to various pins, bodies, or nets on the drawings. If the NO_IO_CHECK property is used as a net property, it applies to all the pins on the net. When used as a body property, it applies to all pins on the body. When used as a pin property, NO_IO_CHECK applies only to the pin to which it is attached.

8.6 DEVICE LOADING CALCULATIONS

Once a design has been expanded into physical components and the interconnections among them are complete, it is necessary to check that loading rules have been obeyed. The loading values are unitless quantities and need not represent any physical values.

The loading for each part is specified in the SCALD library defining the part. The loading for each pin of the part is specified by a property attached to the pin. The property has the following form:

(*low value*, *high value*)

where *low value* is the DC load the pin presents when in the ''0 state'' (the most negative voltage state). The *high value* is the DC load the pin presents when in the ''1 state'' (the most positive voltage state). The actual value is an integer or a real number that specifies the load in some consistent units.

The values used for some Valid libraries (such as the LSTTL library) are the amount of current which an output may source or sink, and the amount of current required to set an input to each of its states. These loading values are specified in mA (Amps x 0.001). By convention, current flowing into a pin is positive and current flowing out of a pin is negative. Some libraries (such as the 100K library) use values which have no physical meaning but instead describe the maximum fan-out for an output pin.

There are two properties used to specify loading: INPUT_LOAD and OUTPUT_LOAD. INPUT_LOAD is used to specify the load a pin presents when it is used as an input or when not driving the signal. An input pin should always have an INPUT_LOAD property. An output pin is given an INPUT_LOAD property whenever that pin can also place an input load on the signal. For instance, a TRI-STATE or an open collector output also presents a load when not driving the signal. This load needs to be considered when calculating the loading of the entire net.

The OUTPUT_LOAD property is used to specify the load presented by a pin when used as an output pin. By definition, the presence of the OUTPUT_LOAD property indicates that the pin is an output pin. If a pin does not have the OUTPUT_LOAD property, it is assumed to be an input pin. When a pin is both an output and an input (e.g., a transceiver pin), both the INPUT_LOAD and OUTPUT_LOAD properties must be present. In addition, the BIDIRECTIONAL property must be used to indicate that the pin is both an input and an output.

The Packager calculates the loading for each net for both logic states ("0 state" and "1 state"). The Packager performs the following checks:

1. For each logic state, the input and output loads must have opposite signs.

2. For each logic state, the absolute value of the smallest output load must be greater than or equal to the absolute value of the total input loads on that net.

Here is an example for a net with four nodes. The loading properties for each node are given as a pair of numbers for the low and high logic states:

			Lo	Hi
Node 1	(Output pin)	OUTPUT_LOAD =	(3.0,	-1.8)
Node 2	(Input pin)	INPUT_LOAD =	(-1.2,	0.2)
Node 3	(Input pin)	INPUT_LOAD =	(-1.2,	0.2)
Node 4	(Bi-di pin)	INPUT_LOAD =	(-1.2,	0.2)
	(Bi-di pin)	OUTPUT_LOAD =	(0.4,	-1.8)

Note that Node 4 is a bidirectional pin that has both an INPUT_LOAD and an OUTPUT_LOAD.

Check 1: For the low logic state, all output load values are positive and all input load values are negative. Likewise, for the high logic state, all output load values are negative and all input load values are positive. In both cases, the input and output loads have opposite signs, satisfying the first Packager load check.

Check 2: For the low state, the smallest output load is 0.4 (Node 4), while the total input load is -3.6 (sum of nodes 2, 3, and 4). Since the absolute value of 0.4 is less than the absolute value of -3.6, this net needs additional output loading and will be flagged as an error. For the high state, the smallest output load value is -1.8, and the total input load is 0.6. Thus, the output loading is sufficient for the high state.

If a net loading error exists and can be fixed by the use of more versions of the net, the Packager will flag the net as having a loading error and will try to generate more versions of the net to correct the error as well as printing an error message. This may in turn cause errors if there are not enough versions of the outputs. (Note that this does not actually change the design.) If a net has a loading error which cannot be fixed by more versions (e.g., an output which cannot drive even a single input), the Packager will flag this error and not try to generate more versions of the net.

LOADING FOR PINS THAT DRIVE OR LOAD ONE STATE ONLY

Some output pins can only drive to one state. For example, an open collector pin can only drive to the "0 state".

For these pins, it is meaningless to specify a loading for the opposite state. Further, the I/O (input/output) and loading checks for the net for the other state should not assume that this pin is an output.

Likewise, some input pins only present a load for one state. Thus the I/O and loading checks for the net for the other state should not assume that this pin is an input. To support this, the Packager allows loading for either the "0 state" or the "1 state" to be specified with an '*' to indicate that the pin does not drive or load the net. For example, the output loading for an open collector pin might be specified as:

OUTPUT_LOAD = (-2.0,*)

indicating that it can drive a 2.0 load in the "0 state" but does not drive the net in the "1 state".

NO_LOAD_CHECK PROPERTY

Device loading calculations may be suppressed on a pin-by-pin or body-by-body basis. The NO_LOAD_CHECK property is used for this purpose. It can be given one of three values as follows:

LOW
This causes the "0 state" loading check to be suppressed. The "1 state" check is performed.

HIGH
This causes the "1 state" loading check to be suppressed. The "0 state" check is performed.

BOTH or TRUE
This causes both the "0 state" and the "1 state" loading checks to be suppressed.

The NO_LOAD_CHECK property can appear as a body property on a library part or on a logical part in the design. When used as a body property, it applies to all pins of the body. This property can also be attached to individual pins of a library part or a logical part in the design. When used as a pin property, NO_LOAD_CHECK applies only to the pin to which it is attached. When the NO_LOAD_CHECK property is attached to a net, it applies to all pins on that net.

UNKNOWN_LOADING PROPERTY

Occasionally there are parts in a design that have pins with unknown or unspecified loading such as the pins of a connector. The Packager makes it possible to include such components in a design without causing net loading or I/O check errors.

The property UNKNOWN_LOADING tells the Packager that loading is unknown, and this inhibits load checks and I/O checks. When UNKNOWN_LOADING is attached as a body property to either a library part or a logical part in a design, the property applies to all pins of the body. When attached to an individual pin on the library part or on a logical part, UNKNOWN_LOADING applies to the entire net to which this pin is attached.

If you attach NO_LOAD_CHECK to a pin with UNKNOWN_LOADING on either the pin or body, load checking will not be suppressed for the entire net, but only for this pin, as specified by the value of NO_LOAD_CHECK. Likewise, attaching NO_IO_CHECK to a pin will only suppress I/O checking for the pin, as specified by the value of NO_IO_CHECK. This mechanism allows you to ''suppress'' the effects of the UNKNOWN_LOADING property on a pin-by-pin basis. This is useful in the case where UNKNOWN_LOADING is attached to the body of a library part.

8.7 TABLES COMPARING NET/LOAD CHECK PROPERTIES

The following tables compare these properties:

- ALLOW_CONNECT
- NO_IO_CHECK
- NO_LOAD_CHECK
- UNKNOWN_LOADING

Note that UNKNOWN_LOADING differs from the other properties: when attached to a pin, it applies to all other pins on the same net.

ALLOW_CONNECT

Used on:	**Applies to:**		
	Pin only	All pins on body	All pins on net
Pin	x		
Body		x	
Net			x

NO_IO_CHECK

Used on:	**Applies to:**		
	Pin only	All pins on body	All pins on net
Pin	x		
Body		x	
Net			x

NO_LOAD_CHECK

Used on:	**Applies to:**		
	Pin only	All pins on body	All pins on net
Pin	x		
Body		x	
Net			x

UNKNOWN_LOADING

Used on:	**Applies to:**		
	Pin only	All pins on body	All pins on net
Pin			x
Body		x	x

SECTION 9
LIBRARIES AND PHYSICAL PART TABLES

The Packager uses information from libraries and physical part tables to determine the physical characteristics of the various parts in a design. The Packager extracts information from these files for each part in the design and places it in the output file PSTCHIP. This output file can then be used by physical design systems.

This section briefly discusses several properties which are important to the Packager and which are attached only to library parts (e.g., PIN_NUMBER, INPUT_LOAD). See the Library reference manual for a more complete discussion.

This section also includes a detailed description of physical part tables, including the use and format of the tables.

9.1 LIBRARY FILES

The CHIPS file contains a description of every physical part in the libraries. It is generated by compiling the library description drawings with the OUTPUT CHIPS directive. The Compiler produces the file CHIPS which is read by the Packager with the LIBRARY_FILE directive in the Packager directives file. The library manager has the responsibility to see that the CHIPS files used by the the designers are up to date. The CHIPS file must be recreated whenever the libraries are modified.

The CHIPS file contains physical information that is entered on the system describing the part. The Packager expects the following information for each part:

1. PIN_NUMBER property for every pin.

2. INPUT_LOAD or OUTPUT_LOAD (or both) properties for every pin.

3. POWER_PINS property for the part.

4. FAMILY property for the part.

Other properties recognized by the Packager but not required:

1. BIDIRECTIONAL pin property if the pin is both an output and an input.

2. UNKNOWN_LOADING pin or body property indicating that device loading is not known.

3. NO_LOAD_CHECK pin or body property used to suppress device loading calculations.

4. NO_IO_CHECK pin or body property used to suppress input and output net checks.

5. WIRE_GATE body property indicating that the body is a phantom wire gate (such as a WIRE-OR).

6. WIRE_GATE_OUTPUT pin property indicating that this is an output pin of a wire gate.

7. OUTPUT_TYPE pin property which specifies whether other outputs can be connected to the pin and what type they must be.

8. ALLOW_CONNECT pin property to permit an output pin to be connected to a net regardless of whether there are other outputs on the net.

9. PHYS_DES_PREFIX body property which specifies the prefix to use for physical part designator creation.

10. PIN_GROUP pin property which specifies whether a pin belongs to a group of swappable pins or not.

11. AUTO_GEN body property to create a default ''library'' for parts that don't exist in the libraries but need to be packaged.

PIN_NUMBER and POWER_PINS PROPERTIES

Library parts must be given PIN_NUMBER properties so the Packager will know how to assign pin numbers, swap sections, etc. The PIN_NUMBER property is attached to each pin of the body (except for bus through pins) and conveys the following information:

- The pin number for the pin.
- How many sections of the part are in a package.
- The pin numbers for each section.

The Packager will print an error message if a pin is found with no PIN_NUMBER property. If a part has multiple sections, the PIN_NUMBER must specify the pin numbers for each section.

Power and ground pin assignments for each part are specified with the POWER_PINS property attached to the part within the libraries. The POWER_PINS property is used to specify both the names of the power rails as well as the pin numbers. The POWER_PINS property only applies to parts found within the libraries and is ignored if found elsewhere.

PIN SWAPPING

The Packager and the section and pin assignment program used by the Graphics Editor recognize the PIN_GROUP property on pins of parts in the CHIPS files. The property is used to assign the logical pins to pin equivalent and swappable groups so that the Packager can perform legal pin swaps.

A swappable group of pins are those pins which are logically equivalent and belong to the same section. This means that if two nets are swapped between two pins which are in a swappable group, the logical function of the circuit is not altered.

A common example of this occurs for the inputs of a NAND gate such as a 74LS00. The two input pins are physically equivalent in terms of loading and propagation delay from input to output. Thus, if the nets to the input pins are swapped, the behavior of the circuit is unchanged.

Any set of pins that are swappable must have the PIN_GROUP attached to them with the same value. Any pin without the PIN_GROUP property is not swappable with other pins. The value of the PIN_GROUP property is not important, only that all pins of a swappable group have the identical value.

If you want to swap pins but the library part does not contain the PIN_GROUP property, use the directive USE_PIN_GROUP OFF.

9.2 PHYSICAL PART TABLES

Physical part tables provide a way to create new part types from a basic part type. For example, you can create many different types of resistors and capacitors from a single basic resistor or capacitor. The various resistor types may have different resistance values, power dissipation, cost, reliability, etc. All of these characteristics can be specified in a physical part table. There is only one library definition for the part, and therefore only one copy of the models. The Packager uses the properties attached to the part to differentiate it from other instances of the same part.

Another use of physical part tables is to attach new body properties to a part type without having to recreate or modify the library files containing the part type definitions. An important use of this capability is the addition of new properties to the libraries for certain interfaces such as SCI-CARDS. These properties describe to the interface the type and shape of each component.

By using several physical part tables, you can change the way part types are handled without changing the library files. This is useful when a design is processed by several different interfaces. The properties for each new interface need not be added to the libraries, but instead are concentrated together into their own special physical part table. You only need to specify the physical part table to be used by the Packager.

You can create a physical part table with any text editor. Since the files are kept in tabular form, they can easily be read and updated.

Below is an example of a physical part table for 1/4-watt resistors. The line numbers on the left are not actually in the file but will be used to describe the format of the table. Note that comments are enclosed in braces, and in this example they precede the element they describe.

```
1.  FILE_TYPE = MULTI_PHYS_TABLE;
2.
3.  { 1/4-watt resistor table }
4.
5.  PART '1/4W RES'
6.
7.  { SCICARDS specific properties }
8.
9.  SCI_PART  = RES1/4W
10. SCI_SHAPE = CR1/4W
11.
12. { table format }
13.
14. : VALUE = PART_NUMBER, COST;
15.
16. { actual table entries for the resistors }
17.
18. 1K   = CB1025, $0.05
19. 1.2K = CB1225, $0.05
20. 1.5K = CB1525, $0.05
21. 2.2K = CB2225, $0.05
22. 2.7K = CB2725, $0.05
23. 3.3K = CB3325, $0.05
24. 3.9K = CB3925, $0.05
```

```
25.  4.7K = CB4725, $0.05
26.  5.6K = CB5625, $0.05
27.  6.8K = CB6825, $0.05
28.  8.2K = CB8225, $0.05
29.
30.  { end of the 1/4W RES entries }
31.
32.  END_PART
33.
34.  { end of the physical part table file }
35.
36.  END.
```

Line 1 is used to start the physical part table file and tells the Packager that the file is a multiple physical part table file. This means it may contain more than one part type.

Blank lines and comments such as lines 2 and 3 are ignored by the Packager to allow you to make the file more readable. The comments are enclosed by '{' and '}'. Comments can cross line boundaries; they cannot be nested.

Line 5 starts the physical part table entries for the '1/4W RES' part type. The part type name must be enclosed by quotes. Lines 9 and 10 indicate that ALL 1/4-watt resistors have the body properties SCI_PART and SCI_SHAPE added to the part type with the values 'RES1/4W' and 'CR1/4W' respectively.

Line 14 describes the format for each line in the table for the 1/4-watt resistor. In this example, the property that may be used to modify the resistor is VALUE and the properties added to the new part types are PART_NUMBER and COST. Another point to be noted is that the separator between the PART_NUMBER and COST properties is a comma. This defines the separator character between the PART_NUMBER and COST values to be a comma within the table that follows.

Lines 18 to 28 are the actual physical part table entries which the Packager searches through to determine the new part types to be created. For example, line 18 specifies that all 1/4-watt resistors that have a VALUE property with a value of '1K' will be assigned to a new part type. This new part type will have the same definition as a 1/4-watt resistor without a VALUE property plus the additional properties PART_NUMBER and COST with the values of 'CB1025' and '$0.05' respectively. If the 1/4-watt resistor had a VALUE of '4.7K' the added PART_NUMBER and COST would instead be 'CB4725' and '$0.05'. Finally, line 32 is used to denote the end of the part table for the part type '1/4W RES' and line 36 denotes the end of the file.

HOW TO USE THE PHYSICAL PART TABLES

Use the PART_TABLE_FILE directive to tell the Packager the names of the files which contain physical part tables. Any number of tables can be specified with this directive. The names can be placed in a list separated by commas or listed individually with separate PART_TABLE_FILE directives. For example, the directive:

```
PART_TABLE_FILE 'res.tab', 'cap.tab';
```

specifies two physical part table files, res.tab and cap.tab, and is equivalent to the directives:

```
PART_TABLE_FILE 'res.tab';
PART_TABLE_FILE 'cap.tab';
```

If a part has a table associated with it, the Packager will read the table format definition line to find the properties that can be used to alter the part. If any of these properties are found on an instance, their values are checked against the entries in the table. If the Packager cannot find an entry in the table for the given values on a part, an error message is generated. You must either change the property values in the drawings or must update the part tables.

The Packager creates a unique library part definition for each entry in the table that matches a use in the drawings. These are summarized as though they were unique physical part types. The associated information from the tables is added to each new library part created and can be used to guide the Packager's execution. For example, attaching a NO_LOAD_CHECK property to a new part type will turn off load checking for all instances of that part type.

To inform other programs, such as DIAL interfaces, that these new library part definitions were created from the physical part tables, the Packager builds a chips file containing all the new part types that are used in the design. This file is written to the logical file PSTCHIP which is bound by default to PSTCHIP.DAT in VMS, pstchip.dat in UNIX, and to PSTCHIP DATA in CMS.

For DIAL interfaces which are run after packaging, you must include the LIBRARY_FILE directive to use the Packager-generated chips file. Thus the interface's directives file might contain the following line:

```
LIBRARY_FILE 'pstchip.dat';
```

NOTE ON SCALE FACTORS

When the Packager searches the physical part table entries, the property values on the instance in a design must exactly match the values defined in the entry. This means that a VALUE property of '1000' on an instance will NOT match an entry with a value of '1K'.

To avoid these problems, it is suggested that you use a consistent set of scale factors for numeric values. One common set of scale factors are those defined for SPICE which are as follows:

```
T   = 1E12
G   = 1E9
MEG = 1E6
K   = 1E3
M   = 1E-3
U   = 1E-6
N   = 1E-9
P   = 1E-12
F   = 1E-15
```

For example, use '1.234K' instead of '1234', and '1MEG' instead of '1000K' or '1000000'.

FORMAT OF THE PHYSICAL PART TABLE

Each physical part table file can contain information for more than one part type and has the following general form:

> FILE_TYPE = MULTI_PHYS_TABLE;
> *part type table*
> *part type table*
> .
> .
> .
> END.

where each *part type table* is the physical part table for a part type. The *part type table* has the form:

> PART '*part name*'
> *part type property list*
> *table format definition*
> *table entries*
> END_PART

where *part name* is the name of the part type being redefined by the table entries, *part type property list* is a list of new properties to be added to the part type, and *table format definition* describes the format of the table which consists of the *table entries.* The end of the part type table is marked with 'END_PART'.

The *part type property list* section of the part tables can be used to add new properties to a part type without having to modify the chip files or library drawings. This is useful if you wish to add properties independent of any set of properties attached to a logical part. Entries in the *part type property list* are of the form:

> *property name* = *property value*

and appears one per line. The *property name* is a standard SCALD property name (e.g., a string of letters, digits, or '_' starting with a letter and no longer than 16 characters).

The *property value* can be any string of characters, and it is terminated by the end of the line. If the value is too long and cannot fit on one line, a tilde (~) may be used as a continuation character. It must appear as the last character in the line. The tilde may appear between any two characters in the line. For example, the line

```
SCI_PART = RES1/4W
```

is equivalent to

```
SCI_PART = RES~
1/4W
```

Notice that multiple spaces are considered to be one space and that leading and trailing spaces about property values are removed. If leading or trailing spaces are required, the property values must be entered with quotes. Thus the line

```
SCI_PART = ' RES1/4W '
```

which defines a SCI_PART value with a leading and trailing space. You may use either single quotes (') or double quotes ("). This allows the use of quotes in the property value by using the other quote character. You can also use the quote character in a quoted string by doubling it when used. For example, the line

```
HOW_ARE_YOU = "I'm OK"
```

is equivalent to the line

```
HOW_ARE_YOU = 'I''m OK'
```

which uses two single quotes within the quoted string to create the quote character (').

The *table format definition* is used to describe the format of each table entry and has the form:

: *instance property list* = *part property list* ;

instance property list is a list of property names that can be attached to an instance of the part. These properties are used to control the selection and customization of the part. For a resistor, this property may be VALUE. This list has the form:

> *property name and attributes*
> or
> *property name and attributes separator char ...*

where if there is more than one *property name and attributes* in the list, they must be separated by a *separator char.*

The *separator char* may be any character but a letter, digit, '_', '=', '(', ')', '{', '}', '~', ':', ';', single quote ('), or double quote (").

Each *property name and attributes* has the form:

> *property name*
> or
> *property name* (*attribute list*)

When present, the *attribute list* describes any special attributes the property may have during processing of the physical part table. The form of the *attribute list* is as follows:

> *attribute*
> or
> *attribute* , *attribute ...*

where if there is more than one *attribute* in the list, they must be separated by commas.

Currently the only *attribute* understood by the Packager is whether a property is optional on an instance of a part. If a property is not optional on a part and is not present on the part, a warning is generated to remind you that the property is missing. To specify that a property is optional, *attribute* has the form:

> OPT
> or
> OPT = '*default value*'

where *default value* is the default value for the property if it is not present on the instance of a part. This default value must appear as a quoted string.

As an example, the definition

```
: VALUE(OPT='1K') = PART_NUMBER;
```

specifies that the VALUE property is optional on the part. If not present on the part, the Packager will assume a default value of '1K' and not generate any warning messages.

The *part property list* is a list of the properties to be associated with the new part type by the Packager. For example, a resistor table may specify the PART_NUMBER property. This property list has the form:

property name
or
property name separator char ...

where if there is more than one property name in the list, the property names must be separated by a *separator char.*

There is no limit to the number of properties that can be specified. The *table format definition* may cross several lines. The semicolon (;) is used to mark the end of the definition.

In the resistor table example, each line is defined to start with a VALUE property followed by an equal sign (=). The next field is the PART_NUMBER property value followed by the COST property value separated by a comma (,). An alternate file with the same information is:

```
FILE_TYPE = MULTI_PHYS_TABLE;
          .
          .
          .

PART '1/4W RES'
          .
          .
```

```
        .

: VALUE = PART_NUMBER COST;
1K   = CB1025  $0.05
1.2K = CB1225  $0.05
1.5K = CB1525  $0.05
        .
        .
        .

END_PART
END.
```

Here the separator has been changed to a space (' '). If the separator is a space, any number of spaces can appear.

The separator characters defined in the format line will be used as the separator characters for property values defined in the table entries. Thus when the separator character was changed from a comma to a space, the comma is no longer special and can be used as part of a property value. Also the ability to change the separator character can be used to make the file more readable. A more creative use of separator characters might be as follows:

```
FILE_TYPE = MULTI_PHYS_TABLE;
                 .
                 .
                 .
{---------------------------------}
: VALUE = PART_NUMBER  | COST
{---------------------------------}
  1K    =    CB1025    | $0.05
  1.2K  =    CB1225    | $0.05
  1.5K  =    CB1525    | $0.05
  2.2K  =    CB2225    | $0.05
{---------------------------------}
                 .
                 .
                 .

END.
```

where the separator character is defined to be a vertical bar (|).

The *table entries* are the actual physical part table entries the Packager searches through to determine the new part types to be created. Each table entry has the form:

> *instance values* = *part type values*
> or
> *instance values* = *part type values* : *new properties*

where the second form is only used when there are additional new properties to be added for the part type created for this table entry. Since a colon (:) is used to separate the last part type property value from any new properties, the last property value must be enclosed in quotes if it contains a colon.

The *new properties* are a list of new part type properties to be added to the new part type created for a particular table entry. The *new properties* have the form:

> *property*
> or
> *property* , *property* ...

where each *property* has the form:

> *property name* = '*property value*'

The property values must be enclosed in quotes. For example if we have the table

```
FILE_TYPE = MULTI_PHYS_TABLE;
PART '1/4W RES'
: VALUE = PART_NUMBER, COST;
1K   = CB1025, $0.05 : TOLERANCE = '5%'
1.2K = CB1225, $0.05
1.5K = CB1525, $0.05
END_PART
END.
```

not only will the part type created for resistors with a VALUE of '1K' have a PART_NUMBER of 'CB1025' and a COST of '_$0.05', but also have a TOLERANCE of '5%'. Resistors with a VALUE of '1.2K' or '1.5K' will NOT have the TOLERANCE property added to the new part type.

Each table entry must each appear on one line. If an entry is too long, we can again use the tilde (~) as a continuation character. For example, the resistor example can be entered as:

```
FILE_TYPE = MULTI_PHYS_TABLE;
          .
          .
          .
PART '1/4W RES'
          .
          .
          .
: VALUE = PART_NUMBER,
          COST;
1K   = CB1025,~
      $0.05
1.2K = CB1225,~
      $0.05
1.5K = CB1525,~
      $0.05
          .
          .
          .
END_PART
END.
```

If more than one property is specified in the *instance property list*, the AND of the values is used. For example,

```
FILE_TYPE = MULTI_PHYS_TABLE;
PART '1/4W RES'
: VALUE, TOLERANCE = PART_NUMBER COST;
1K,   5%= CB1025    $0.05
1K,   1%= CB1021    $0.50
1.2K, 5%= CB1225    $0.05
1.2K, 1%= CB1221    $0.50
END_PART
END.
```

Note that both the VALUE and TOLERANCE properties must match the values as specified in the table before the property entry can be found. In this case, changing the TOLERANCE property on a 1K resistor causes a different part to be selected (with a corresponding change in cost).

MODIFIED PART TYPES IN PHYSICAL PART TABLES

The Packager normally generates the new part type names for modified part types (subtypes) by appending a dash (-) and an integer to the part type name. Examples of subtype names for the part "RESISTOR" are RESISTOR-1, RESISTOR-2, and RESISTOR-3.

These subtype part type names are not guaranteed to be associated with the same parts from one run to another. For example, the Packager might assign the name RESISTOR-1 to the part 12P in the first run, and then assign the name RESISTOR-2 to the same part in another run.

You can assign your own subtype name in the physical part table by attaching it to the property name. In this way the subtype name remains the same from one Packager run to the next. Here is an example of a physical part table with subtype names:

```
line  PART  'RESISTOR'

1   :VALUE,  TOLERANCE      =  PART_NUMBER  COST
2     1K        2% (1K)     =     1285      $.50
```

3	2.3K	1% (2.3K)	=	1300	$.50
4	1K	5% (1K, 5%)	=	1024	$.24
5	5K	1% (!)	=	1000	$.43
6	1K	3% (!)	=	1028	$.24
7	1K	4%	=	1028	$.24

The subtype names (suffixes) are enclosed in parentheses. The suffixes are of two types: explicit or implicit. Implicit subtypes are denoted by an exclamation point (!). They cause the Packager to append the property values to the part type name. The Packager places commas (,) between property values. For example, line 5 above would result in the subtype RESISTOR-5K,1%, and line 6 would be RESISTOR-1K,3%.

Explicit subtypes appear directly within the parentheses. For example, line 3 would result in the subtype RESISTOR-2.3K, while line 4 would be RESISTOR-1K,5%.

The characters which are allowed within a suffix are as follows:

- a through z
- A through Z
- 0 through 9
- ,
- $
- %
- #
- &
- *

- +
- _
- .

The length of the part type and the suffix together cannot exceed 255 characters or the length of a legal part type name as defined by default or directive. If the name is longer than this limit, it is truncated. The PART_TYPE_LENGTH directive controls the part type name length limit.

If no suffix is specified, the Packager creates a numeric suffix and appends it to the part type name. For example, line 7 might be RESISTOR-2.

SECTION 10
TIMESAVERS/TROUBLESHOOTING

This section covers several timesaving techniques for advanced Packager users and also offers some pointers for troubleshooting a design.

Here are some timesaving techniques:

1. Only generate the output files you really need. If it is early in the design and you don't need to keep consistent assignments from one run to the next, then omit the state files. Likewise, if you are not doing back annotation, omit the back annotation file. The report files and cross reference files are another set of optional files which you may not need for all Packager runs. You can control the generation of these files through the directives OUTPUT, REPORT, and USE_STATE_FILES. (Note: be sure to turn State files on when tracking design information.)

2. You can make some changes through feedback files instead of going through the Graphics Editor and recompiling the design. However, be sure to back annotate and recompile at a later point to keep the schematic and the Packager output synchronized.

3. If you have not introduced any components of a new part-type into the design, and if there are no changes to the physical part tables or libraries for the design, you can use the output file PSTCHIP as the input library file for the design. This saves time because the Packager does not need to search through the various libraries and physical part tables to extract the parts for the design. Be careful when doing this, because if there are any new part types, the Packager run will fail. For example, if your original design contains only the components LS00 and LS04, you

can use PSTCHIP as the input library file for the next Packager run only if the new design contains only LS00 and LS04 parts.

To use PSTCHIP as your new input library, rename the file, as in this example

```
%mv pstchip.dat mylib.prt
```

and then enter the file name in the LIBRARY_FILE directive:

```
LIBRARY_FILE 'mylib.prt';
```

Then run the Packager.

When you are troubleshooting a design, it is often helpful to back annotate and print a copy of the annotated schematic. Also, there are several output files which are useful for finding problems:

- PSTLST - the listing file, which shows error messages.
- PSTLCHG - shows the logical parts which were added or deleted since the last Packager run.
- PSTPCHG - shows all physical parts which were added or deleted since the last Packager run.
- PSTBCHG - shows all bindings which were added or deleted since the last run.
- PSTXREF - contains three cross reference files: local part, global signals, and global part.
- PSTRPRT - contains two reports showing the remaining spare sections and the number of packages of each physical part required for this design.

For more information, see the detailed descriptions of Packager output files in Section 13.

SECTION 11
ADVANCED TOPICS

This section is not yet available. It will appear in later editions of the Packager manual.

SECTION 12
PACKAGER DIRECTIVES

Directives are commands to the Packager program. You enter these commands in the text file packager.cmd. The file consists of a series of directives followed by "end." Here is an example:

```
library_file '/u0/lib/lsttl/lsttl.prt';
warnings on;
oversights on;
end.
```

Each directive is followed by a semicolon (;). The directives can appear in any order. You can enter directives in upper or lower case characters. However, if you reference UNIX files, be sure to put the file name in lower case. Comments are allowed but must be enclosed in curly braces ({ }). The file can be free-form: the Packager ignores end-of-line and multiple spaces.

Note that you must always specify a library in packager.cmd through either the LIBRARY_FILE or LIBRARY directive. Otherwise, the Packager will not be able to find the physical characteristics of the logical parts in the design.

This section describes the format and function of all Packager directives. The directives appear in alphabetical order. The syntax illustrations for the directives use the following conventions:

[ ]	Square brackets enclose optional entries
name	Names in italics stand for specific entries which you supply or pick from a list
...	Ellipses indicate that you can repeat the previous entry

text	Items in standard text should be copied exactly (e.g., directive titles, semicolons)

Here is an example to illustrate the syntax conventions:

report *rptname [,rptname]*;

rptname

spares	List of spare physical sections available for use
partsummary	Summary of all the physical parts used in the design

In the above example, *rptname* stands for the entries ''spares'' and ''partsummary''. Since the second occurrence of *rptname* is optional, both of the following are syntactically correct:

```
REPORT SPARES;
REPORT PARTSUMMARY, SPARES;
```

The syntax illustrations use lowercase characters because they are easier to read. The directives embedded in the text are shown in upper case for clarity of expression. However, the directives file itself is not case-sensitive.

ANNOTATE

Use this directive to specify the information to be included in the back annotation file (PSTBACK). The Graphics Editor uses this file to mark physical part designators, pin numbers, and/or physical net names on the schematic. This directive controls only the data to be included in the file; the OUTPUT directive controls whether the file is produced. The form of this directive is:

annotate *option [,option] [,option]*;

option

body	Back annotate physical part designators

pin	Back annotate physical pin numbers
net	Back annotate physical net names

Default: When you omit this directive, the Packager generates back annotation for BODY and PIN.

Notes: The NET option is for scalar nets only and requires the synonym file from compilation (CMPSYN). Also, if the ANNOTATE directive appears more than once in the directives file, the Packager ignores all but the last one.

DOCUMENT_ERRORS

Use this directive to control the printing of detailed error messages in the listing file (PSTLST). This does not control the printing of the original, brief error message. The detailed error messages are short paragraphs which appear after the list of errors and give more information about possible causes of the error. This information is also available at the back of this manual in Section 15.

document_errors on;	print detailed error messages in listing
document_errors off;	do not print detailed error messages

Default: This directive is ON if omitted.

FEEDBACK_ORDER

This directive specifies the feedback files and the order in which the Packager will process them. The files are processed in the order they are listed in the directive. (See Section 6 for a detailed discussion of feedback files.)

feedback_order [-v] *filetype* [*,filetype*] ... ;

filetype

part_trans	Physical part designator transformations file (PSTPRTX) - to change part names
section_trans	Physical section reallocation file (PSTSECX) - to change part names and section connections
feedback_netlist	Feedback net list (PSTFNET) - to change pin connections, section connections, and part names
net_trans	Physical net name transformations file (PSTNETX) - to change net names

The "-v" (verbose) option generates additional error information for certain errors found during feedback processing. For example, error #149 "Match not found for feedback section" generates a list of the closest partial matches for the entry. Normally, only the first three partial matches are listed. The "-v" option outputs all partial matches in the listing file.

Default: When you omit this directive, the Packager does not perform feedback processing.

Notes: This directive can appear only once in the directives file. Also, state files must be generated for feedback; otherwise, the changes cannot be saved. Therefore, you must also include the directive USE_STATE_FILES ON. The state files must also currently exist; that is, the Packager run prior to the feedback run must create state files.

FILTER_PROPERTY

This directive specifies properties to be omitted from the expanded net and part lists. Other files are not affected. You can list any number of properties, and you can enter this directive as many times as needed in the directives file.

This directive is useful for filtering out properties from library part descriptions which are irrelevant to the design.

filter_property *property [,property]* ... ;

property	Any property specified for this design through the Graphics Editor.

Default: No properties are filtered.

Note: A related directive is PASS_PROPERTY, which is performed before FILTER_PROPERTY. In this way, properties can be explicitly passed and then filtered.

FREE_GROUPING

This directive controls the assignment of parts lacking the GROUP property.

free_grouping off;	Parts without the GROUP property are in the "default" group and cannot be assigned to the same package as parts in other groups.
free_grouping on;	Parts without the GROUP property are "free" and can be assigned to the same package as parts with the GROUP property.

Default: This directive is OFF if omitted.

HARD_GROUPING

This directive temporarily nullifies the effect of the GROUP property for the entire design so that you can overwrite this property. Use this directive to move a section from one group's physical part to another group's physical part during feedback. Note that inadvertent feedback errors can be very damaging in these circumstances, so use the directive with caution. Also, be sure to back annotate to record these changes on the schematic, and then recompile the design before packaging it again.

hard_grouping on;	The GROUP property is in effect
hard_grouping off;	The GROUP property is nullified

Default: This directive is ON if omitted.

HARD_LOC_SEC

This directive temporarily nullifies the effect of the LOCATION and SECTION properties for the entire design so that you can overwrite them. Use this directive if you want to change LOCATION or SECTION assignments during a feedback run. Note that inadvertent feedback errors can be very damaging in these circumstances, so use the directive with caution. Also, be sure to back annotate to record these changes on the schematic, and then recompile the design before packaging it again.

hard_loc_sec on;	LOCATION and SECTION properties are in effect
hard_loc_sec off;	LOCATION and SECTION properties can be overwritten

Default: This directive is ON if omitted.

INCLUDE_IO_LIST

Use this directive to mark interface signals with the IO_NET property. Interface signals are those attached to a FLAG body. The type of FLAG body determines the value of IO_NET (INPUT, OUTPUT, or BIDIRECTIONAL).

include_io_list on;	output the IO_NET property for interface signals marked with the FLAG body
include_io_list off;	do not output the IO_NET property for interface signals

Default: This directive is OFF if omitted.

LIBRARY

This directive is similar to LIBRARY_FILE, but you can use the short version of the library name instead of the full path name. Use the directive to specify the libraries containing physical information for parts in the design. The directive can appear more than once, but a library file cannot be listed more than once.

library '*shortname*';

shortname	The short name assigned to a full path name.

Example: library 'lsttl', 'sttl';

For example, the short name for /u0/lib/lsttl/lsttl.prt is lsttl. Most standard Valid libraries have short names that you can use. These are the same names you use with the LIB command in the Graphics Editor. Be sure to enclose the short file name in single quotes. Note that you can use LIBRARY and LIBRARY_FILE together in the same directives file.

Default: None. You must use LIBRARY or LIBRARY_FILE at least once in the directives file, since the Packager must know the part libraries.

LIBRARY_FILE

This directive specifies the libraries containing physical information for parts in the design. The directive can appear more than once. A library file cannot be listed more than once. Note that you can use the short version of a library file name with the directive LIBRARY.

library_file *'file' [,'file'] ...* ;

file	The full path name of the library file. The path name must be enclosed in single quotes.

Example: library_file 'u0/lib/lsttl/lsttl.lib';

Default: None. You must use LIBRARY or LIBRARY_FILE at least once in the directives file, since the Packager must know the part libraries.

MAX_ERRORS

This directive specifies the maximum number of errors allowed before the Packager halts. When this happens, the Packager prints a message and terminates with a summary of the execution.

max_errors *n*;

n	The number of errors allowed before the Packager halts.

Default: If you omit this directive, the Packager terminates after 1000 errors.

NET_NAME_LENGTH

This directive controls the maximum length for physical net names generated by the Packager. If the maximum length is too short (e.g., 6), the Packager will be unable to generate unique names for all physical net names and will print an error message.

net_name_length *length*;

length	Maximum number of characters for a physical net name

Default: If you omit this directive, the maximum length is set to 24 characters.

OUTPUT

Use this directive to suppress particular output files. The OUTPUT directive can appear more than once in the directives file. For example, you can use OUTPUT; to suppress all the output files and then use OUTPUT ABC; to enable only file ABC. The OUTPUT directive controls only the files listed below under *outfile.*

output all;	Enable all output files
output -all;	Suppress all output files
output -sample;	Suppress output file ''sample''
output;	Suppress all output files

output *outfile [,outfile]* ... ;

outfile

chipsfile	Output the chips file to PSTCHIPS

expandednetlist	Output the expanded net list to file PSTXNET
expandedpartlist	Output the expanded part list to file PSTXPART
logicalchanges	Output the summary of changes in logical parts in the design to file PSTLCHG
physicalchanges	Output physical changes to file PSTPCHG
bindingchanges	Output binding changes to file PSTBCHG
crossreferences	Output all cross references to file PSTXREF
localpartxref	Output local part cross references to file PSTXREF
globalsignalxref	Output global signal cross references to file PSTXREF
globalpartxref	Output global part cross references to file PSTXREF
backannotation	Output back annotation to file PSTBACK

Default: If you omit this directive, the Packager produces all the above files except the cross reference files.

OVERSIGHTS

Use this directive to control the display of oversight messages. An oversight is a condition which is more serious than a warning but not as serious as an error. An oversight should be corrected, but the design will probably work without the correction. The total number of oversights is always reported at the end of the Packager listing file (PSTLST). This directive controls the printing of detailed oversight messages.

oversights on;	display all oversight messages on the Packager listing file
oversights off;	omit oversight messages

Default: Display oversight messages.

Note: Use the SUPPRESS directive to turn off individual oversight or warning messages.

PART_NAME_LENGTH

This directive controls the maximum length for physical part names generated by the Packager. If the maximum length is too short (e.g., 4 characters), the Packager will be unable to generate unique names for all physical parts and will print an error message.

part_name_length *length*;

length	maximum number of characters

Default: If you omit this directive, the maximum length is set to 16 characters.

PART_TABLE_FILE

This directive specifies the files containing physical part tables to be referenced by the Packager. There is no limit on the number of files. You can use this directive more than once in the directives file, but a file name can be listed only once. (For more information on physical part tables, see Section 9.)

part_table_file *'filename' [, 'filename'] ...* ;

filename	The full path name of the physical part table. Be sure to enclose the path name in single quotes.

Default: You must use this directive to reference any physical part tables.

PART_TYPE_LENGTH

This directive limits the length of the names of part types that are modified by physical part tables. (For more information on physical part tables, see Section 9.) This directive controls the length of subtype suffixes which can be attached to part type names.

part_type_length *n*;

n	The maximum part type length, where *n* is an integer between 0 and 255.

Default: 24

Note: The maximum part type length must be an integer between 0 and 255, and it must be greater than or equal to the length of the longest part type name used in the design.

PASS_PROPERTY

This directive specifies properties to be passed through to the expanded net and part lists. Other files are not affected. You can list any number of properties, and you can enter this directive as many times as needed in the directives file. Use this directive to explicitly select properties which are to be passed; those not named will be excluded from the expanded net and part lists.

pass_property *property [,property] ...* ;

property	Any property specified for this design through the Graphics Editor.

Default: All properties are passed.

Note: A related directive is FILTER_PROPERTY, which is performed after PASS_PROPERTY.

PRINT_PIN_LIST

This directive generates a file required by the Drawing Flattener. The file (PSTPIN) contains the PIN_NUMBER of each PIN_DEF of the PART_TYPES used. The Packager marks with "$S" each pin that:

- Is not a common pin
- Is sizeable (contained in more than one section)
- Is not a vectored pin

print_pin_list on;	Generate the file PSTPIN for the Drawing Flattener.
print_pin_list off;	Do not generate the file PSTPIN

Default: The file PSTPIN is not generated.

REPORT

This directive specifies the user reports to be included in the output file PSTRPRT.

report -all;	omit all reports
report all;	include all reports
report;	omit all reports

report *rptname [,rptname]*;

rptname

spares	List of spare physical sections available for use
partsummary	Summary of all the physical parts used in the design

Default: If you omit this directive, all reports are produced.

SUPPRESS

Use this directive to suppress specific warnings and oversight messages. You cannot suppress error messages. The design conventions assumed by the Packager are conservative and rigorous. You may choose to design in a more liberal style and may want to ignore certain messages.

suppress *n [,n]* ... ;

n	The number of the warning or oversight message to be suppressed

Note: You can suppress all warnings with the WARNINGS directive and all oversights with the OVERSIGHTS directive.

USE_PIN_GROUP

This directive is useful if you want to do a pin swap and are using a library without the PIN_GROUP property in the pin description. Normally, the Packager only swaps pins with the same PIN_GROUP property value. The Packager will ignore this requirement if you specify USE_PIN_GROUP OFF.

use_pin_group off;	Ignore pin_group property for pin swaps.
use_pin_group on;	Check pin_group property for pin swaps.

Default: If you omit this directive, it defaults to ON.

USE_STATE_FILES

This directive controls the use of state files.

use_state_files on;	Use state files if present and generate new state files.
use_state_files off;	Do not use or generate any state files.

Default: If you omit this directive, the Packager will use and generate state files.

Note: If you are using feedback files, you must use state files as well.

WARNINGS

Use this directive to control the display of warning messages. A warning is a condition which is less severe than an oversight or error. You should correct it, but the design will work without corrections. The total number of warnings is always reported at the end of the Packager listing file (PSTLST). This directive controls the printing of detailed warning messages.

warnings on;	Display all warning messages in the Packager listing file.

warnings off;	Display no warning messages.

Default: Display warning messages.

Note: Use the SUPPRESS directive to turn off individual oversight or warning messages.

SECTION 13
PACKAGER OUTPUT FILES

This section contains detailed format descriptions of Packager output files. The Packager produces the following output files:

User Files
- Listing File (PSTLST)
- Log File (PSTLOG)
- Cross References (PSTXREF)
- Logical Changes Summary (PSTLCHG)
- Binding Changes List (PSTBCHG)
- Physical Changes List (PSTPCHG)
- Reports File (PSTRPRT)

Net/Part Files
- Expanded Net List (PSTXNET)
- Expanded Parts List (PSTXPRT)
- Chips File (PSTCHIP)

Back Annotation File (PSTBACK)

State Files for Later Packager Runs
- Logical Signal Name to Physical Net Name Binding (PSTSIGB)
- Logical to Physical Part Designator Binding (PSTPRTB)
- State File (PSTSTAT)
- Pin Swap File (PSTPSWP)

Within the section, these output files are described in the order they appear above. However, the following files are not described:

Listing File
Log File
Chips File
Back Annotation File

State Files

The Listing File contains error information for the user, and the Log File contains error information for use by Valid personnel. The Listing File is described in Section 2.

The following files are created by the Packager for use by other programs. DO NOT EDIT THESE FILES. Since they are created in a specific format, they will not work properly if you change them.

- The Chips File contains information about parts used in the design. This information is extracted from library files and combined with property data from Physical Part Tables (if present). The Chips File is used by physical design systems.

- The Back Annotation File is used by the Graphics Editor.

- The State Files are used by the Packager to maintain consistent assignments from one run to the next.

GENERAL NOTES ON OUTPUT FILE FORMAT

The Valid canonical signal name form is used for all logical signal names and pin names. The assertion and name portion of the signal are in quotes. No bit lists appear. The Valid canonical syntax is:

'[-] *name' <subscript>*

where '-' is required for low asserted signals. The subscript appears within angle brackets (< >).

Comments may be placed in the expanded net and part lists if enclosed in '{' and '}'. A comment may appear anywhere a space may appear. Comments may cross line boundaries but may not be nested.

If an item is too long to fit on a line (80 characters), it must be broken into more than one line. A tilde (~) is used as a continuation character to indicate that the current item is continued on the next line. A line break may appear between any two characters in the file. A tilde is only significant if it occurs at the end of the line.

Some of the Packager output files have very complex formats. For the descriptions in this section, the file structure is broken down into simpler parts in a hierarchical manner. The overall structure of the file is shown first, followed by the main individual parts. Some parts have substructures that are also discussed in detail. Repetitive elements in a file are indicated by ellipses (. . .).

13.1 CROSS REFERENCE FILES (PSTXREF)

There are many questions that need to be answered during the design, test, debugging, and construction of a design. The Cross Reference files are intended to directly or indirectly answer many of these questions. There are two basic cross reference types. The local cross reference is sorted by logical information and relates to a single drawing. The global cross reference is sorted by physical information and relates to the design as a whole.

There are three cross references contained within this output file:

Local Part Cross Reference
Global Signal Cross Reference
Global Part Cross Reference

You can omit any or all of these cross references through the OUTPUT directive. (See ''Controlling Cross Reference Generation'' below for more details.) These cross references appear in PSTXREF in the order listed above.

This section is organized as follows:

Overview of the three cross references
Explanation of path properties and path elements
Controlling cross reference generation through

the OUTPUT directive
Detailed format descriptions for the three cross references

LOCAL PART CROSS REFERENCE OVERVIEW

The Local Part Cross Reference file contains information about the logical parts in the design and the physical assignments given them. It is produced for each drawing in the design and lists all of the logical parts that are found in the drawing. It identifies a logical part by giving its name and the PATH property attached to it. (See below for an explanation of PATH elements.) Given a logical part in a drawing, the designer can easily find the corresponding entry in this cross reference since it is ordered by logical part designator. If there is more than one instance of a particular logical part within the drawing, the specific logical part can be identified by its PATH property. The physical part to which the logical part has been assigned is also given. If a logical part has been assigned to more than one physical part due to SIZE or TIMES replication, the physical part is given for each SIZE and TIMES replicated logical part.

The logical and physical signal names are given for each pin on the logical part. The designer can see the logical signal name in the drawing and this cross reference gives the physical net name assigned to it. The Global Signal Cross Reference (which is indexed by physical net name) can be checked to find all of the other logical parts on the net. If there is more than one signal name for a signal in the drawing (because of synonyms or interface signals), this cross reference can be used to determine which name the system uses to refer to the signal.

GLOBAL SIGNAL CROSS REFERENCE OVERVIEW

The Global Signal Cross Reference contains information about each net in the entire design. It is sorted by physical net name. For each physical net, the logical signal name is shown. This is the same logical signal name that appears in the drawings. The loading on the net is given for both the 0-state and the 1-state. This loading is the sum of all of the input loads on the net.

Each node on the net appears with its associated physical part name and pin number. This information makes it possible to trace a net in the physical design. The part type and pin name are also given. To make it possible to find the node in the drawings, the logical part corresponding to a particular pin is given by specifying a PATH element and the drawing in which the logical part can be found. (See below for an explanation of PATH elements.)

GLOBAL PART CROSS REFERENCE OVERVIEW

The Global Part Cross Reference contains the same information as the Local Part Cross Reference except that it is sorted by physical part rather than logical part and refers to the entire design rather than a single drawing.

The part type is given for each physical part along with the pins on the part. For each pin, the physical and logical signal names are given. The logical part corresponding to the particular pin is given by the PATH element of the part and the drawing where the part appears. To find the part, get the drawing that is referenced (it refers to a specific page), find the part on the drawing, and make sure it has the PATH property given in the PATH element. (See below for an explanation of PATH elements.) Note that the Global Signal Cross Reference shows the parts to which a physical net is connected.

PATH PROPERTIES AND PATH ELEMENTS

This section explains the syntax associated with path properties and path elements so that you can interpret the output files correctly. PATH information is used in several places to specify a particular logical component. The term "PATH property" refers to the PATH property attached to a logical component. The term "PATH element" refers to a specific SIZE and/or TIMES replicated logical part. For example, examine the following part:

```
         PATH=2P
         SIZE=32B
         TIMES=2T
        +--------+
        |        |
      - |D      Q| -
        |        |
        |        |
        |        |
        |  LS374 |
        |        |
        |        |
        |        |
        |        |
      - |CLK     |
        |        |
        +--------+
```

The LS374 (an octal register) has been given three properties. The PATH property serves to identify this LS374 on a particular drawing, the SIZE property causes this LS374 to be 32 bits wide, and the TIMES property causes two versions of each output to be created (thereby doubling the number of components).

The PATH property for the LS374 in the drawing is 2P. This property describes a particular component as it appears in a drawing. A component in a drawing may be described in terms of many logical components. The LS374 above represents 64 (SIZE 32 x TIMES 2) logical components. The SCALD III language has a consistent method for naming each of the logical components. This name is called the "PATH element." A PATH element has the form:

*PATH property # SIZE index * TIMES index*

PATH property is the PATH property attached to the component in the drawing. As SIZE expansion is performed, a *SIZE index* is used to number each of the logical components. The LS374 above is given SIZE indices that run from 0 to 31 (since the SIZE property value is 32). The SIZE index value is appended to the PATH property separated by a '#'. As TIMES expansion is performed, a

TIMES index is used to number each of the logical components. The LS374 above has TIMES indices that run from 0 to 1 (since the TIMES property value is 2). The TIMES index value is appended to the PATH element after the SIZE index, separated by an '*'. PATH elements for the LS374 above are:

```
2P
2P*1
2P#1
2P#1*1
2P#2
2P#2*1
2P#3
2P#3*1
    .
    .
    .
2P#31
2P#31*1
```

Note that the SIZE or TIMES index value is omitted when it is zero. This prevents SIZE (#0) and TIMES (*0) values on parts that have no SIZE or TIMES properties.

CONTROLLING CROSS REFERENCE GENERATION

The cross references are generated by the Packager under the direction of the OUTPUT directive. There are two ways to control the cross references: they may all be turned on together, or they may be turned on individually. By default, the Packager generates all of the cross references. To direct the Packager to generate all of the cross references, use the directive:

```
OUTPUT CROSSREFERENCES;
```

This causes ALL of the cross references to be generated. Each cross reference may be individually selected as well. The following directive generates the Local Part Cross Reference:

```
        OUTPUT LOCALPARTXREF;
```

All of the cross references are output to the file PSTXREF. If more than one cross reference is output, they are separated by page ejects. All local cross references are separated by page ejects since each local cross reference refers to a single drawing page. The file may be split up into individual cross references by breaking at the page ejects. The cross references always appear in the same order in the PSTXREF file independent of the order that they are specified in the OUTPUT directives. The cross references are all output assuming at least 132 characters are permitted in a line. There is no provision to specify the width of the output file.

LOCAL PART CROSS REFERENCE FILE FORMAT

A Local Part Cross Reference is produced for each drawing in the design. It is a list of all of the logical parts in the drawing sorted by logical part name. The following information is given for each part:

Logical part name
Part's PATH property
Physical designator for part
List of pins of the part with:
    Pin number for each pin
    Physical net name connected to each pin
    Pin name for each pin
    Logical signal connected to each pin

The general form for a cross reference entry is as follows:

```
logical part   PATH property   physical part

  pin number   physical net   pin name   logical net
  pin number   physical net   pin name   logical net
  pin number   physical net   pin name   logical net
                    .
                    .
                    .
```

An entry in such a cross reference for the logical part 100166 might appear as follows:

```
100166   29P   U18
   1     OPB2   B<2>    OP B<2>
   2     OPB1   B<1>    OP B<1>
   3     OPB0   B<0>    OP B<0>
   4     LTL    B>A     -LT
   5     EQ     -A=B    EQ
   8     GT     A>B     GT
   9     OPA0   A<0>    OP A<0>
  10     OPA1   A<1>    OP A<1>
  11     OPA2   A<2>    OP A<2>
  12     OPA3   A<3>    OP A<3>
  13     A0     A<4>    0
  14     A0     A<5>    0
  15     A0     A<6>    0
  16     A0     A<7>    0
  17     A0     A<8>    0
  19     A0     B<8>    0
  20     A0     B<7>    0
  21     A0     B<6>    0
  22     A0     B<5>    0
  23     A0     B<4>    0
  24     OPB3   B<3>    OP B<3>
```

The first line gives the logical part name (100166), the part's PATH property (29P), and the name of the physical part where this logical part was placed (U18).

The rest of the lines in the entry show each pin of the part. The first line shows the pin number (1), the physical net name (OPB2), the logical pin name for the pin (B<2>), and the logical signal name connected to the pin (OP B<2>).

If the logical part is given a SIZE or TIMES property, the cross reference entry summarizes the common portions of the logical part and then lists each of the SIZE and/or TIMES replicated sections. For example, the logical part 100145 with SIZE=4 might appear as follows:

```
100145   26P   SIZE=4   Summary of common pins:
   1    READADRB2      AR<2>   READ ADR B<2>
   2    READADRB1      AR<1>   READ ADR B<1>
   3    READADRB0      AR<0>   READ ADR B<0>
   14   C2L            -OE0    -C2
   15   C2L            -OE1    -C2
   16   C1L            -WE0    -C1
   17   C1L            -WE1    -C1
   19   MR             MR      MR
   20   WRITEADRAB0    AW<0>   WRITE ADR AB<0>
   21   WRITEADRAB1    AW<1>   WRITE ADR AB<1>
   22   WRITEADRAB2    AW<2>   WRITE ADR AB<2>
   23   WRITEADRAB3    AW<3>   WRITE ADR AB<3>
   24   READADRB3      AR<3>   READ ADR B<3>

   Section:   26P#3   U1
      4    OPB3      Q<3>   OP B<3>
      13   RESULT3   D<3>   RESULT<3>

   Section:   26P#2   U1
      5    OPB2      Q<2>   OP B<2>
      12   RESULT2   D<2>   RESULT<2>

   Section:   26P#1   U1
      8    OPB1      Q<1>   OP B<1>
      11   RESULT1   D<1>   RESULT<1>

   Section:   26P   U1
      9    OPB0      Q<0>   OP B<0>
      10   RESULT0   D<0>   RESULT<0>
```

The first line contains the logical part name (100145), the part's PATH property (26P), and a SIZE value specification showing the number of SIZE and/or TIMES replicated parts (SIZE=4). After the first line, the common pins of the part are shown in the same format as the 100166 example shown above:

pin number   physical net   pin name   logical net

For example:

```
1  READADRB2   AR<2>  READ ADR B<2>
```

The SIZE and/or TIMES replicated sections of the logical part appear after the common pins summary. The form for each SIZE and/or TIMES replicated section is:

Section: *PATH element* *physical part designator*
pin list

The *PATH element* consists of the PATH property followed by the SIZE replicated index (#1, #2, #3, ...) and the TIMES replicated index (*1, *2, *3, ...). The name of the physical part to which the logical section is assigned is given last. The *pin list* is of the same form as the *pin list* in the common pins summary; only pins that are unique to the specified section are listed.

GLOBAL SIGNAL CROSS REFERENCE FORMAT

A Global Signal Cross Reference is produced for the entire design. It consists of a list of all of the signals in the design sorted by physical net name. The following information is given for each net (signal) in the cross reference:

- Physical net name
- Low state input load
- High state input load
- Logical signal name of the net
- List of nodes on the net with:
  - Physical part designator
  - Pin number on the part
  - Pin name of the pin
  - Part's part type
  - Corresponding logical part
  - Drawing on which this signal resides

The general form for an entry in this cross reference is:

```
physical net   net loading   logical signal

          node description
          node description
               .
               .
               .
```

An entry in the cross reference might appear as follows:

```
READADRB2   3.0   -3.0   READ ADR B<2>
U1   1    AR<2>   100145   26P#3    A1.LOGIC.1.1
                           26P#2    A1.LOGIC.1.1
                           26P#1    A1.LOGIC.1.1
                           26P      A1.LOGIC.1.1
U2   24   Q<26>   100150   33P#26   A1.LOGIC.1.1
```

The first line of the above entry shows the *physical net* (READADRB2). Following the net name is the total load on the net presented from all the inputs on the net. Both the 0-state (3.0) and the 1-state (-3.0) loading are shown. The last entry on the first line is the logical signal name for the net (READ ADR B<2>). Following the first line is a list of all of the nodes on the net. Each *node description* has the form:

physical part designator pin number pin name part type PATH drawing

In the above example, the *physical part designator* is U1, the *pin number* is 1, the *pin name* is AR<2>, and the *part type* is 100145. The *PATH element* is (26P#3), and the *drawing* in which the part appears is A1.LOGIC.1.1. The PATH element is made up of the PATH property on the logical part and the SIZE and TIMES replication values for this expanded instance. Note that the physical part, pin, and part type information are not listed for the next three entries. Instead of repeating identical information, the cross reference leaves it blank. This is intended to make the cross reference easier to read.

GLOBAL PART CROSS REFERENCE FILE FORMAT

A Global Part Cross Reference is produced for the entire design. It is a list of all of the physical parts in the design sorted by physical part designator. The following information is given for each part:

Physical part name
Part type
List of the nodes on the part with:
  Pin number of each pin
  Physical net connected to the pin
  Logical signal name connected to the pin
  PATH element for the logical part
  Drawing on which the logical part is found

The general form for an entry in this cross reference is:

physical part designator   part type

pin number   physical net   logical net   PATH   drawing
pin number   physical net   logical net   PATH   drawing
pin number   physical net   logical net   PATH   drawing

.
.
.

An entry in such a cross reference for the part 100145 might appear as follows:

```
U1   100145
1    READADRB2    READ ADR B<2>    26P#3   A1.LOGIC.1.1
2    READADRB1    READ ADR B<1>    26P#3   A1.LOGIC.1.1
3    READADRB0    READ ADR B<0>    26P#3   A1.LOGIC.1.1
4    OPB3         OP B<3>          26P#3   A1.LOGIC.1.1
5    OPB2         OP B<2>          26P#2   A1.LOGIC.1.1
8    OPB1         OP B<1>          26P#1   A1.LOGIC.1.1
9    OPB0         OP B<0>          26P     A1.LOGIC.1.1
10   RESULT0      RESULT<0>        26P     A1.LOGIC.1.1
11   RESULT1      RESULT<1>        26P#1   A1.LOGIC.1.1
12   RESULT2      RESULT<2>        26P#2   A1.LOGIC.1.1
13   RESULT3      RESULT<3>        26P#3   A1.LOGIC.1.1
14   C2L          -C2              26P#3   A1.LOGIC.1.1
15   C2L          -C2              26P#3   A1.LOGIC.1.1
```

```
16 C1L          -C1                26P#3  A1.LOGIC.1.1
17 C1L          -C1                26P#3  A1.LOGIC.1.1
19 MR           MR                 26P#3  A1.LOGIC.1.1
20 WRITEADRAB0  WRITE ADR AB<0>    26P#3  A1.LOGIC.1.1
21 WRITEADRAB1  WRITE ADR AB<1>    26P#3  A1.LOGIC.1.1
22 WRITEADRAB2  WRITE ADR AB<2>    26P#3  A1.LOGIC.1.1
23 WRITEADRAB3  WRITE ADR AB<3>    26P#3  A1.LOGIC.1.1
24 READADRB3    READ ADR B<3>      26P#3  A1.LOGIC.1.1
```

The first line of the entry shows the *physical part designator* which is U1 in this example. Following that is the *part type*, 100145. The following lines show the pins of the part in numerical order. The pin number, physical net name, and logical net name are given for each pin. The last two entries on the line describe the logical part that corresponds to the pin. The logical part is described by giving its PATH element and the drawing on which the logical part appears.

The first entry in the pin list above is:

```
1   READADRB2   READ ADR B<2>   26P#3   A1.LOGIC.1.1
```

which shows pin number 1 connected to the physical net READADRB2 which is also the logical signal READ ADR B<2>. The logical part corresponding to this pin has a PATH property of 26P with a SIZE replication index of 3. This part appears in the drawing A1.LOGIC.1.1.

13.2 LOGICAL CHANGES SUMMARY (PSTLCHG)

The Logical Changes Summary shows the logical parts that were added to or deleted from the design since the last run of the Packager. The format of the logical part description is:

```
(logical part path name) part type ;
     #SIZE *TIMES
```

An example is:

```
(SPT .8SIP2P) 8SIP;
     #0*0
```

For more information on the format of the *logical part path name*, see the SCALD III Language reference manual.

13.3 BINDING CHANGES LIST (PSTBCHG)

The Binding Changes List shows all bindings that were changed or deleted during the Packager run. A binding is a mapping of a logical part to its allocated physical section. Changed bindings are listed in the form:

(*logical part path name*) *part type* *#SIZE *TIMES*
IS ASSIGNED TO *physical part* SECTION *section number*

For example:

(SPT .8SIP1P) 8SIP #1*0 IS ASSIGNED TO U3 SECTION 1

For more information on the format of the *logical part path name*, see the SCALD III Language reference manual.

13.4 PHYSICAL CHANGES LIST (PSTPCHG)

The Physical Changes List shows all physical part designators that were added to or deleted from the design during the Packager run.

13.5 REPORTS FILE (PSTRPRT)

The Reports File consists of two reports: the Spares List and the Part Summary. The Spares List shows all the spare physical sections in the design. These are sections that have not been allocated to a logical part. Spares are listed in the form:

physical part designator - pin number

For example, "U3-3" indicates that the section containing pin 3 of physical part U3 has not been allocated.

The Part Summary is an alphabetical list of abbreviations for all the part types used in the design. It also shows the number of physical parts of each part type in the design. Here is a sample report:

```
- PART SUMMARY -

TADD          2
TDFF          2
TINV          1

Total         5
```

13.6 THE EXPANDED NET LIST (PSTXNET)

The Expanded Part List and Expanded Net List are input files to physical design systems. These files contain a complete description of the design and can be converted into a form compatible with some target physical design system. The files are organized by physical information.

The Expanded Net List is ordered by physical net name and contains all net properties and the logical to physical binding of nets and nodes. The form of the Expanded Net List is as follows:

```
FILE_TYPE=EXPANDEDNETLIST;

net description;
  .
  .
  .
END.
```

The *net description* is of the form:

```
NET_NAME
physical net  logical net  version :

net property
        .
        .
        .  ;
node description
        .
        .
        .  ;
```

where NET_NAME is used to mark the beginning of a net entry. The *physical net* is a quoted string of letters and digits. The *logical net* is the signal in Valid canonical syntax. The *version* is an optional field of the form

> * *version number*

where *version number* specifies which version of the TIMES replicated signal matches the physical net. Versions of the signal are numbered from 0 to number_of_versions - 1. If *version number* is 0, then it is omitted. A colon (:) is used to mark the end of the logical net name. Each *net property* has the form:

> *property name = 'property value'*

Each element in the list of properties is separated by a comma (,). The last element in the list is followed by a semicolon (;). This list may cross several lines. If there are no properties, only a semicolon appears.

There are *node descriptions* for each of the nodes on the net, including node properties. Each node description has the form

```
NODE_NAME
physical part designator   pin number

logical node description
        .
        .
        .
```

where *physical part designator* is the name of the physical part. The name is a string of letters and digits. *pin number* is the pin number on that part. There is a *logical node description* for each logical node corresponding to the physical node. The *logical node description* has the form

```
logical designator   bit   version :   pin name :
  node property
        .
        .
        .  ;
```

where *logical designator* is the name of the logical part corresponding to the physical part. It is a quoted string of characters.

The optional field *bit* specifies which bit of a SIZE-replicated component matches the physical section. Bits of the component are numbered from 0 to number_of_bits - 1. If *bit* is 0, then it is omitted.

The *version* is an optional field described above for *net description*. The *pin name* is the logical name of the pin corresponding to the pin number given. It has Valid canonical syntax (see the paragraph ''General Notes on Output File Format'' at the beginning of this section).

A colon (:) is used to mark the end of the *pin name* and the start of the node properties. The *node properties* are output in the same manner as the net properties described above.

Here is an example of an Expanded Net List:

```
FILE_TYPE=EXPANDEDNETLIST;
NET_NAME
 'DATA2'                { physical net DATA2 }
 -'DATA2'<2>:           { logical net name }
  LENGTH='2',           { property of the net }
  BREADTH='4';          { property of the net }
                        { first node on the net }
NODE_NAME
 U31 2                  { physical part designator
                          and pin number }
 '(TEST1 ABD2P#2 3.4GN2P TF1.14P#1 SDP1P TBB23P ~
GDP2P .74.5P)74LS74': 'Q'<0>: { note line break }
                        { previous line contains
                          the logical designator
                          and pin name }
  INPUT='23';           { node property }
                        { second node on the net }
NODE_NAME
 U11 3                  { physical node }
 '(TEST1 .00.12P)74LS00'#2*1: { bit 2, version 1 }
  'Y'<0>:               { first logical node }
  ;                     { no node property }
 '(TEST1 .00.13P)74LS00':
  'Y'<0>:               { second logical node }
  ;                     { no node property }
NET_NAME                { the next net }
      .
      .                 { and so on }
      .
END.
```

The Expanded Net List is designed to be easily read. Net descriptions are marked by the NET_NAME keyword and thus cannot be confused with a net name since physical net names cannot contain the underscore character (_). The net name entry is always terminated by a semicolon (;) which comes at the end of the net property list.

The physical net name is always followed by the logical name which has a restricted form making it easy to interpret. A colon is used to mark the end of the logical net name so that there is no chance for confusion with properties.

After the net names and properties are the node entries. Each of these is marked by the NODE_NAME keyword and terminated with a semicolon (;) which falls at the end of the list of node properties. The logical designator is followed by a colon (:) to mark the start of the pin name. The pin name is followed by a colon to mark the beginning of the node properties. Line boundaries are not significant; they should not be used to determine where one item begins and the other ends. The file is totally free form.

13.7 THE EXPANDED PART LIST (PSTXPRT)

The Expanded Part List is ordered by physical part designator and contains all the information known by the Packager for each of the parts. This includes all the properties and the logical to physical bindings of parts.

The form of the Expanded Part List is as follows:

```
FILE_TYPE=EXPANDEDPARTLIST;
DIRECTIVES
  ROOT_DRAWING='root drawing name';
  COMPILE_TIME='compilation time';
  POST_TIME='packaging time';
  global design properties
END_DIRECTIVES;

part description;
          .
          .
          .
END.
```

root drawing name
  Name of the root drawing that was compiled.
compilation time
  The time and date of the compilation of the design.
packaging time
  The time and date when the design was packaged.
global design properties
  A list of design-wide properties specified in the

directives section of the Compiler command file.

Each entry in *global design properties* is of the form

property name = '*property value*';

where *property name* is the name of the property and *property value* is the value of the property (which is enclosed by quotes).

The *list of parts* is of the form

```
PART_NAME
physical part designator   part type name : ;

logical part description
          .
          .
          .   ;
```

where PART_NAME is used to mark the beginning of a part entry. The *logical part description* shows all the logical parts that are allocated to the physical part. It has the form

```
SECTION_NUMBER   section number
logical designator   bit   version :

  logical part property
          .
          .
          .   ;
```

where *section number* is a number indicating which section of the physical part matches the logical part. The *logical designator* is the name of a logical part. The *bit* and *version* are the optional fields specified above for the expanded net list. A colon (:) is used to mark the end of the logical part designator. The *logical part properties* are the properties of the logical part. These properties are read from the Compiler expansion file and come from your drawings. The properties in the list have the form:

property name = '*property value*'

Each element in the list of properties is separated by a comma (,). The last element in the list is followed by a semicolon (;). This list may cross several lines. If there are no properties, only a semicolon appears.

Here is an example of an Expanded Part List:

```
FILE_TYPE=EXPANDEDPARTLIST;
DIRECTIVES
  ROOT_DRAWING='RISC';
  COMPILE_TIME=' COMPILATION ON date/time';
  POST_TIME='19-OCT-1983 ';
END_DIRECTIVES;
PART_NAME
U31                    { physical part U31 }
 '74LS00':;
SECTION_NUMBER 1
                       { first logical part }
 '(TEST1 ABD2P#2 3.4GN2P TF1.14P#1 SDP1P TBB23P
GDP2P .74.5P)':        { note line break }
                       { previous line contains
                         the logical part name }
  SIZE='1';            { logical part property }
SECTION_NUMBER 2
        { second logical part in the 74LS00 }
 '(TEST1 .00.2P)'#2*1:; { no properties }
SECTION_NUMBER 3
                       { third logical part }
 '(TEST1 ABD2P#2 .00.5P)':
  SIZE='2',
  TIMES='1';
SECTION_NUMBER 4
                       { fourth logical part }
 '(TEST1 FGH1P DD2P .00.11P)':;
PART_NAME              { the next part }
      .
      .                { and so on }
      .
END.
```

The Expanded Part List is designed to be easily read. Part descriptions are marked by the PART_NAME keyword and thus cannot be confused with a part name, since physical part designators cannot contain the underscore character (_). The physical part designator is always terminated by a semicolon (;) which comes at the end of the part type property list. The physical part designator is always followed by the part type name which is in quotes to make it easy to identify.

A colon and semicolon are used to mark the end of the part type name and the beginning of the physical sections. Each of these physical sections is terminated with a semicolon (;) which falls at the end of the list of logical part properties. Each physical section begins with SECTION_NUMBER which makes it easy to recognize. The logical designator is in quotes making it easy to interpret. It may be followed by SIZE and TIMES replication information. Next is a colon (:) to mark the beginning of the logical part properties. Line boundaries are not significant; they should not be used to determine where one item begins and other ends. The file is totally free form.

SECTION 14
GLOSSARY

The following glossary is included in the hope that it will make this chapter easier to understand.

Binding

A mapping of a logical part to its allocated physical section.

Expanded Part

Each logical part is expanded to the number of its SIZE and/or TIMES property. An expanded part is a unit of this expansion. For example, a logical part with SIZE=4 will have four expanded parts.

Fatal Error

This is a class of errors. When an error in this class is detected the Packager does not alter any state files or produce any output files other than the error listings. Execution continues after the detection of a fatal error in order to find any other errors that may be present.

Logical Part

A body on the design drawing created in the Graphics Editor. It may have SIZE and TIMES properties attached that are used by the Packager to generate several logical parts (sections).

Logical Part Designator

The name given to a particular occurrence of a logical part. This name is assigned by the Compiler and consists of the path name for the part and the logical part type. The path name is augmented by the Packager when replicating parts with SIZE properties or creating new versions because of TIMES properties. Logical and physical part names have no correspondence.

Logical Pin Designator

A logical part designator and a logical pin name separated by a space.

Logical Pin Names

The name given a pin of a body drawing.

Net

A connection among nodes.

Node

A pin connected to a net.

Part Type

The part type is assigned in the libraries as the PART_NAME property. If the PART_NAME property is not present, the logical part type is the same as the drawing name.

Path Name

A unique name assigned by the Compiler to each logical part (e.g., 1P, 3P, 8P).

Path Property

The name the Packager assigns to each expanded part. This name is derived from the path name.

Phantom Gate

A ''phantom'' library body representing a wire-gate which connects versions of the signals used only in the gating function.

Physical Name String

A sequence of characters consisting only of letters, digits, or '_'.

Physical Net Name

Each net has a logical signal name (assigned by the Compiler and derived from the drawings) and a corresponding name used by the physical system. The physical net name is the name used by the physical system to refer to the net.

Physical Part

A physical chip on a board -- a ''package''.

Physical Part Designator

The name given to an instance of a physical part. Each physical part in a design has a unique physical part designator that can be assigned manually by the designer or automatically by the Packager. Logical and physical part names do not have a one-to-one correspondence.

Physical Part Type

The name of a physical part as assigned in the SCALD library. For example, a package of TTL NAND gates may have the physical part type 74LS00. This name may in many cases be a generic part name or it may be an internal part name.

Physical Pin Designator

Consists of a physical part designator and a physical pin name separated by a space.

Physical Pin Name

The name given a pin on a physical part. By convention this name is a number or an identifier (not more than 16 characters). It is specified by the PIN_NUMBER property on the library component describing the physical part type.

Section

A section is a place to assign a logical expanded part to a physical part. Each expanded part is placed in a section of a physical part.

Signal Name

A name assigned to a net or a portion of a net.

Wire-Gate

The connection of two or more outputs to the same signal.

Wire-Tie

Same as wire-gate.

SECTION 15
PACKAGER ERROR MESSAGES

There are three classes of problems detected by the Packager:

1. Errors

   An error is a problem that must be fixed before further progress can be made. For example, a missing drawing would be considered an error since some portion of the design is missing. Some errors are fatal and cause the Packager to stop. For example, if an input file is missing, the Packager will generate an error message and stop.

2. Oversights

   An oversight is a less severe problem than an error. For example, if a library is specified twice in the directives file, the Packager generates an oversight message. The design will work; however, since the file is specified twice, there may be confusion and the design may be difficult to work with. In general, oversights may be ignored for a while, but should be fixed eventually.

3. Warnings

   A warning is a problem that is very minor. For example, if a directive is specified more than once, the Packager generates a warning message. There is no reason to remedy this condition because the Packager will always produce the same output. The user may elect to correct warning conditions.

15.1 FORMAT OF MESSAGES

The Packager's error, oversight, and warning messages have the following format:

> #*n* ERROR(*m*): *message*
> #*n* OVERSIGHT(*m*): *message*
> #*n* WARNING(*m*): *message*

where *n* indicates how many of the particular class of messages have occurred so far, *m* is the message code number, and *message* is the text of the message. For example,

```
#27 ERROR(22): String length exceeded
```

is the 27th error found in this run of the Packager. In addition, the error code is 22, which indicates ''String length exceeded''. This means that some string (a quoted sequence of characters) is longer than the allowable maximum of 255.

Following the message are several lines describing where the error was detected, the part, the pin on the part, etc. This information is intended to specify the location of the error as accurately as possible to simplify finding and correcting the problem.

The error, oversight, and warning messages are counted separately. At the end of compilation, the total number of each is reported. Here is an example:

```
47 errors detected
No oversights detected
6 warnings detected
```

15.2 DIRECTIVES AFFECTING ERROR MESSAGES

You can use the DOCUMENT_ERRORS directive to control the printing of detailed error messages in the listing file (PSTLST). This does not control the printing of the original, brief error message. The detailed error messages are short paragraphs which appear after the list of errors and give more information about possible causes of the error. These are the same paragraphs which appear in this section. The directive DOCUMENT_ERRORS ON prints detailed error messages in the listing. Use DOCUMENT_ERRORS OFF to omit the detailed messages. If you omit this directive, it defaults to ON.

There are several other directives which affect the printing of oversights and warning messages. The WARNINGS directive controls the display of warnings messages. If you use WARNINGS ON, warning messages are printed as usual. WARNINGS OFF omits all warning messages. The OVERSIGHTS directive works in the same way for oversight messages. To suppress an individual warning or oversight message, use the SUPPRESS directive with the number(s) of the specific message(s) to be omitted. The SUPPRESS directive cannot be used for errors; only for warnings or oversights.

If you want the Packager to halt after a certain number of errors, use the directive MAX_ERRORS. The Packager normally halts after encountering 1000 errors, but you can use MAX_ERRORS to change this to a different number.

15.3 SPECIAL MESSAGE REPORTING

Some problems are detected while reading input files. These usually are syntax problems, typically occurring when text was mistyped. For these errors, the Packager prints the text being read and points with the character '^' to the position in the text where the problem was detected. For example, given the signal name:

```
DATA < 0 [the first bit} .. 31 {the last bit} >
```

an error #20 will be displayed as follows:

```
DATA < 0 [the first bit} .. 31 {the last bit} >
                       ^
#1 ERROR(20): Unmatched closing comment character
```

The error really occurs earlier where '[' was used instead of

'{'. It is usually impossible for the Packager to accurately determine the true position of the error; the pointer is always at the position where the error was detected.

15.4 SUMMARY OF MESSAGES BY NUMBER

The remainder of this section consists of the Packager error messages. The messages are listed in numerical order. Each listing consists of the error, oversight, or warning message itself followed by an explanatory paragraph.

ERROR #1: Expected identifier

The Packager expected an identifier (a string of letters, digits, or '_' starting with a letter) and found some other data. Identifiers are used as names in properties, text macros, and as operands for the Packager directives. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #2: Expected =

The Packager expected an equal (=) and found some other data. Equals are used in many places: between property names and values, in expressions, and in the FILE_TYPE specification at the beginning of data files. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #3: Expected [

The Packager expected an opening bracket ([) and found some other data. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #4: Expected]

The Packager expected a closing bracket (]) and found some other data. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #5: Expected a constant

The Packager expected a constant (sequence of numeric characters) and found some other data. The Packager prints the input line with a pointer to the position in the line where the problem was detected. Constants are expected in many places. For example, the bit offset on a SIZE expanded part is a constant.

ERROR #6: Expected subrange specifier

The Packager expected a subrange specifier (e.g., the ".." in <15..0>) and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #7: Expected)

The Packager expected a right parenthesis ()) and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #8: Expected ,

The Packager expected a comma (,) and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #9: Expected *

The Packager expected an asterisk (*) and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #10: Expected <

The Packager is expecting a left-angle bracket (<) and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #11: Expected >

The Packager expected a right-angle bracket (>) and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #12: Expected ;

The Packager expected a semicolon (;) and found some other character. The Packager prints the input line where the problem was detected. The Packager continues to read the input despite the missing semicolon.

ERROR #13: Expected :

The Packager expected a colon (:) and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #14: Unexpected symbol in integer expression

The Packager found something unexpected while reading an expression (e.g., a selection expression or a bit subscript). The Packager prints the input line with a pointer to the position in the line where the problem was detected. The Packager expected one of the following:

1. A constant
2. An expression in parentheses, e.g. (2+ 3).
3. NOT followed by an item from this list.

ERROR #15: Expected (

The Packager expected a left parenthesis (() and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #16: Bit value invalid

The Packager read a bit subscript and found an illegal bit value. The Packager prints the input line with a pointer to the position in the line where the problem was detected. The bit value is also printed. Bit values are invalid if they are negative or are greater than the largest allowed bit number. Since the largest allowed bit number is 2**31-1 (2147483647), this error usually means that the bit value is negative. This error is most likely to occur when specifying a bit with an expression. An example is DATA<SIZE-2..0> when SIZE=1.

ERROR #17: Non-unique PATH_NAME in CMPEXP.DAT

The Package found the same PATH_NAME on more than one logical part. Each logical part should have a PATH_NAME that identifies it uniquely throughout the design. PATH_NAMEs are produced by the Compiler. This error is often caused by manual edits to the Compiler expansion file. To fix this error, compile the design again before re-packaging.

ERROR #18: Unused.

ERROR #19: Unused.

ERROR #20: Unmatched closing comment character

The Packager encountered a closing comment character (}) without a previous matching starting comment character ({). The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #21: Unused.

ERROR #22: String length exceeded

The Packager read a string exceeding the maximum allowed length. Strings are limited to 255 characters. The Packager prints the input line with a pointer to the position in the line where the problem was detected. The string is truncated at the current position (pointed to by the Packager) and the Packager reads until it finds the closing quote or the end of the input line. Make the string shorter.

ERROR #23: Illegal character found

The Packager found an illegal character in an input file. All non-printing characters except TAB are illegal. The Packager prints the input line with a pointer to the position in the line where the problem was detected. Remove the character.

ERROR #24: Expression value overflow

The Packager evaluated an expression whose value overflowed. The Packager prints the input line with a pointer to the position in the line where the problem was detected. An overflow does not cause the Packager to abort; it assigns the value zero (0) to the result (unless it knows a more reasonable value) and continues.

ERROR #25: Division by zero

The Packager detected division by zero during evaluation of an expression. The Packager prints the input line with a pointer to the position in the line where the problem was detected. Division by zero does not cause the Packager to abort; it skips the division and continues with the packaging.

ERROR #26: Unused.

ERROR #27: Unused.

ERROR #28: Unused.

ERROR #29: Unused.

ERROR #30: Unexpected symbol in bit subscript

The Packager found unexpected characters in a bit subscript. The Packager prints the input line with a pointer to the position in the line where the problem was detected. The symbols expected by the Packager in a bit subscript are:

1. A subrange symbol (..).
2. A colon (:) specifying a bit step.
3. A comma (,) specifying the start of the next element in a bit list.

4. A right angle-bracket (>) indicating the end of the subscript.

ERROR #31: Unused.

ERROR #32: Non-printing ASCII character found

The Packager detected a non-printing character while reading from an input file. This is illegal. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #33: Expected a string

The Packager expected a string (a quoted sequence of printing characters) and found some other data. The Packager prints the input line with a pointer to the position in the line where the problem was detected. Strings are expected in the following places, among others:

1. In signal names: a signal property must have a property value specified as a string.
2. In the Packager directives: the name of the library file for the design must be specified as a string.

ERROR #34: Comment not closed before end of input

The Packager did not find the end of a comment before the end of the file. A comment is started with the "{" character and ended with the "}" character. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #35: Unused.

ERROR #36: Unused.

ERROR #37: Expected .

The Packager expected a period (.) and found some other character. The Packager prints the input line with a pointer to the position in the line where the problem was detected. This oversight is most commonly caused by omitting the '.' following the END at the end of the directives or text file.

ERROR #38: Unused.

ERROR #39: Undefined identifier in expression

The Packager found an undefined identifier (a string of letters, digits, or '_' starting with a letter) in an expression. Identifiers are used as names in properties and text macros. The Packager prints the input line with a pointer to the position in the line where the problem was detected. If the identifier is supposed to be a defined text macro, check the DEFINE bodies to make sure it was correctly defined. If it was supposed to be a parameter of the body, check the body definition to make sure that it was correctly defined there.

ERROR #40: Expected END

The Packager reached what it expected to be the end of a file and did not find an END statement. The most common places where an END is required are at the end of the directives file, a text macro, and property attribute files. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #41: Identifier length exceeded

The Packager encountered an identifier (a string of letters, digits, or '_' starting with a letter) with more than 16 characters. Identifiers are used as names in properties and text macros. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

The Packager ignores the rest of the identifier and continues.

ERROR #42: Unused.

ERROR #43: Unused.

ERROR #44: Unused.

ERROR #45: Unused.

ERROR #46: Unused.

ERROR #47: Unused.

ERROR #48: Unused.

ERROR #49: Unused.

WARNING #50: This file name was already specified

A file name was specified more than once in a directive. Most likely, a library file name was given more than once in the directives file.

ERROR #51: Unknown directive

The Packager found an unknown directive in the directives file. The Packager prints the input line with a pointer to the position in the line where the problem was detected. This error will not normally prevent the Packager from reading the rest of the directives.

ERROR #52: Invalid specification for directive

The Packager found an invalid operand while reading the directives file. The Packager prints the input line with a pointer to the position in the line where the problem was detected. Several of the Packager directives require an operand (e.g., WARNINGS requires either ON or OFF as its operand). If the operand for any of these directives is not what the Packager expected, this error is generated. See the Packager directives documentation for a complete description of the directives.

ERROR #53: Input line exceeds maximum length

The Packager tried to read a line longer than its 255-character input buffer. The first 255 characters of the input line are printed with a pointer to the last character. The input line must be broken into shorter lines before the Packager can read it. Although the Packager's input buffer is 255 characters long, we recommend that input lines be no longer than 80 characters for ease of viewing on a terminal.

ERROR #54: Unexpected error for suppression

The parameter for the directive SUPPRESS is not in the legal range. This value should be an integer in the range 0 to 250.

ERROR #55: This error cannot be suppressed

The parameter of the SUPPRESS directive corresponds to an error that cannot be suppressed. It is only possible to suppress OVERSIGHTS and WARNINGS, which indicate user flaws that do not prevent the packaging of the design. More serious errors cannot be suppressed.

ERROR #56: Unused.

ERROR #57: End of input before end of expression

The Packager found an incomplete expression in an input file. The Packager prints the input line with a pointer to the position in the line where the problem was detected. A blank line printed by the Packager means that the input line was null. This may occur when an empty string was given as a parameter to a text macro that expects an integer parameter.

ERROR #58: Extraneous characters at end of expression

The Packager found extra characters at the end of an expression. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #59: Unused.

ERROR #60: Number of errors must be > 0

The directive MAX_ERRORS was issued with an illegal parameter. The parameter for this directive must be a positive integer.

ERROR #61: Radix must be in range 2 to 16

The Packager encountered a constant signal name with an illegal radix specification. The Packager prints the input line with a pointer to the position in the line where the problem was detected. Signal constants may be specified in any base from 2 to 16. All other bases are illegal. The default base is 2.

WARNING #62: Physical name length must be > 1 and < 255

The directive PART_NAME_LENGTH was issued with an illegal parameter. The parameter for this directive must be an integer between 1 and 255.

ERROR #63: Unused.

ERROR #64: HAS_FIXED_SIZE property conflicts with SIZE property.

The values of the HAS_FIXED_SIZE property and the SIZE property are unequal. The HAS_FIXED_SIZE property is normally added only to body drawings, and not to the components in the design. The value of the HAS_FIXED_SIZE property implies a SIZE property of the same value. It is an error to attach both of these properties with different values to the same component.

ERROR #65: HAS_FIXED_SIZE property value exceeds # of sections in package.

The value of the HAS_FIXED_SIZE property is greater than the number of available sections on the physical package. The name of the offending logical part is printed along with the error message. Look up the library description for this part to find the number of sections available on the package. This information could be determined from the PIN_NUMBER property. For example, the following property

```
PIN_NUMBER='(1,3,9,13)'
```

indicates that there are four sections on this part. The HAS_FIXED_SIZE property of any instance of this part in the design should have a value of four or less.

ERROR #66: Unused.

ERROR #67: Unused.

ERROR #68: Unused.

ERROR #69: Unused.

ERROR #70: Non-contiguous bit subscripts for pin

The bit subscripts of a vectored pin are not contiguous. The part type and the pin name of the part in error are printed after the error message. Change the library entry for this part type so that the bits of each vectored pin are in consecutive order. This error could also be caused by conflicting signal names for the same net. Note that the Packager currently does not support the segmentation of a vectored pin. For example, the pin BUS<15..0> may not appear in the design as BUS<15..8> and BUS<7..0>.

ERROR #71: Unused.

ERROR #72: Unknown signal syntax specification

An unknown signal syntax directive is found in the directives block of the Compiler expansion file. This error will not occur if the directives block of this file is not manually edited, since the file is machine-generated. If the error occurs, recompile the design before packaging again.

ERROR #73: Signal syntax element found twice

A signal syntax directive appears twice in the directives block of the Compiler expansion file. If the Compiler expansion file has not been manually edited, check for a duplicate syntax entry in the directives block.

ERROR #74: Every syntax MUST have a name portion

The signal syntax in the directives block of the Compiler expansion file has no NAME portion. This portion of the syntax is needed in order to name the signal. This error probably resulted from manual edits to the Compiler expansion file. Try compiling the design again before re-packaging.

ERROR #75: Every syntax MUST have a subscript

The signal syntax in the directives block in the Compiler expansion file does not contain a SUBSCRIPT portion. This portion of the syntax is required to process vectored signals. This error probably resulted from manual edits to the Compiler expansion file. Try compiling the design again before re-packaging.

ERROR #76: Illegal form for signal syntax

The directives block of the Compiler expansion file is not in the format expected by the Packager. This error occurs when the Packager does not find one of the following items in the beginning of the signal syntax specification:

NAME
NEGATION
ASSERTION

This error is most likely caused by user edits to the Compiler expansion file. Try compiling the design again before re-packaging.

ERROR #77: Symbol must be one character

This is an error in the directives block of the Compiler expansion file. The error occurred because the value for some configuration parameter is more than one character long. The values for the following parameters may be no more than one character long:

LOW_ASSERTION
HIGH_ASSERTION
NEGATION
NAME_PREFIX
GENERAL_PREFIX
CONCATENATION

This error is most probably caused by user edits to the Compiler expansion file. Try compiling the design again before re-packaging.

ERROR #78: This symbol cannot be used here

A parameter in the directives block has a forbidden symbol for a value. The forbidden symbols include the following characters:

1. ;
2. <
3. >
4. The digits 0 through 9
5. #

This error is probably due to manual edits to the Compiler expansion file. Try compiling the design again before re-packaging.

ERROR #79: Subrange symbol must be < .. > or < : >

The SUBRANGE directive in the directives block of the Compiler expansion file has a value other than a colon (:) or a double period (..). These are the only values understood by the Packager for the directive SUBRANGE.

ERROR #80: No pins found on part in drawing

A part in the drawing has no pins. This is unacceptable to the Packager. The part must either be deleted from the design or changed.

ERROR #81: No Pins found on library part

A part in the design contains no pins in its library description. This is unacceptable to the Packager, and the library description for the part must be changed to include some pins.

ERROR #82: Cannot open Master Library file.

The file specified by the directive MASTER_LIBRARY does not exist, or it cannot be opened due to permission problems.

ERROR #83: Subtype suffix is illegal

The Packager detected an illegal user-specified subtype suffix in the physical part tables. When the Packager creates a new part type from the properties in the physical part table, it makes a new name for this "subtype" by appending to the original part type name a '-' and a suffix. The user may specify this suffix in the physical part table. The subtype suffix must be given following the instance property value, and must be enclosed in parentheses. For example:

```
PART 'RESISTOR'
  :VALUE, TOLERANCE                  = COST
  (string)      1K  ,   5%(1K)       = $1
  (!)           1K  ,   3%(!)        = $2
```

The subtype suffix can be given in either of the two formats shown above: (string) or (!). The subtype resulting from the first line of this example is 'RESISTOR-1K'. In the second format (line 2), the Packager appends to the original part type name all instance property values for the new part type, separated by commas. The subtype for line two of the above example is 'RESISTOR-1K,3%'. For further details, see the Packager reference chapter section on physical part tables.

WARNING #84: Changing HARD_GROUPING from OFF to ON causes split of parts

The Packager separated logical parts of different GROUPs that were previously assigned to the same physical package. Logical parts having different GROUP property values are normally assigned to separate packages. They can be forced into the same package, however, by doing FEEDBACK with the directive HARD_GROUPING OFF. To keep these parts in the same package, the user should back annotate, then compile before running the Packager again.

Without backannotation and a compilation, the Packager would split the parts according to the GROUP properties if HARD_GROUPING is ON. This message does not indicate an error; it informs the user that a change in assignments was made due to HARD_GROUPING ON.

ERROR #85: Expected FILE_TYPE specification

The Packager started reading an input file and did not find a FILE_TYPE specification. All SCALD system data files, except the directives file, are identified with a FILE_TYPE specification at the beginning. The Packager prints the input line with a pointer to the position in the line where the problem was detected. It ignores the rest of the offending file. To fix this error, check the given file to make sure that it is the correct file, and then add the proper FILE_TYPE specification.

ERROR #86: File is not of the correct type

The Packager found an input file with the wrong FILE_TYPE specification. All SCALD system data files, except the directives file, must be identified with a FILE_TYPE specifier which allows the programs to check the validity of input data. The Packager prints the input line with a pointer to the position in the line where the problem was detected, and it ignores the rest of the offending file. To fix this error, make sure that it is the correct file, and then change its FILE_TYPE specification.

ERROR #87: Unused.

ERROR #88: Cannot assign part to feedback section.

The Packager cannot assign a part to the feedback location specified in the feedback files. This is probably due to some conflict with the GROUP or LOCATION_CLASS property. Be sure that parts of different GROUPs are not being fed back to the same package unless the directive HARD_GROUPING is OFF.

ERROR #89: String not closed before the end of line

The Packager found that a string (a quoted sequence of characters) did not have a closing quote before the end of the current line. This most commonly occurs when a quote is accidentally placed in a signal name. The Packager prints the input line with a pointer to the position in the line where the problem was detected. Check the line to make sure that the string is correct.

ERROR #90: Vector PIN_NUMBER < pin's width

The size of a vectored pin in a library description for a part is less than the width of the pin. The size of the vectored pin is the number of physical pins sharing the same pin name on the package. The width of the pin is the number of bits in this pin. Consider the following example:

```
      PIN          PIN_NUMBER property
   'A'<3..0>          '(1,4,9)'
  (width = 4)      (3 physical pins)
```

This will cause an error because the pin is defined as having a width of four while there are only three physical pins available.

Change the library so that the bit width of the vectored pin matches its number of physical pins.

ERROR #91: Vector PIN_NUMBER > pin's width

The size of a vectored pin in a library description for a part is greater than the width of the pin. The size of the vectored pin is the number of physical pins sharing the same pin name on the package. The width of the pin is the number of bits in this pin. Consider the following example:

```
      PIN          PIN_NUMBER property
   'A'<2..0>         '(1,4,9,13)'
  (width = 3)      (4 physical pins)
```

This will cause an error because the pin is defined as having a width of three while there are four physical pins available.

Change the library so that the bit width of the vectored pin matches its number of physical pins.

ERROR #92: Invalid operand of SUPPRESS directive

The operand of the SUPPRESS directive is an error code. This is invalid since only warnings or oversights may be suppressed. The Packager prints the input line with a pointer to the position in the line where the problem was detected.

ERROR #93: Expected directory file name

The Packager did not find a directory file name where it expected one. The Packager prints the input line with a pointer to the position in the line where the problem was detected. This error may occur for three reasons:

1. The file name may simply have been forgotten.
2. The file name may not have been quoted.
3. A comma was found after a file name with no other file names following.

ERROR #94: PIN_NUMBER values are all zero

All of the pin numbers for a part in the library are zero. This is obviously illegal since every pin on the part must have at least one valid pin number for mapping. The pin numbers for the library part must be changed.

ERROR #95: Feeding back multiple nets for a common pin

There is an error in the Feedback Netlist File (PSTFNET). The Packager has detected multiple net names fed back to a common pin.

In a physical package having multiple sections sharing a common pin, this shared pin connects to a single net. When giving feedback information, all common pins of the same PIN_NUMBER must have the same net name.

ERROR #96: Cannot split HAS_FIXED_SIZE part

A logical part with the HAS_FIXED_SIZE property has been forced into different packages. A entire HAS_FIXED_SIZE part must stay together as a unit when it is allocated to physical packages. A part with HAS_FIXED_SIZE = 4, for instance, is equivalent to one with SIZE = 4, except that all four expanded parts must be assigned to the same package. This error is most likely caused by user-generated feedback which assigned the expanded parts of a logical part to different packages.

ERROR #97: Expansion file not for Packager

The Compiler expansion file does not have the correct file type. The file type is found in the first line of the file. For a Compiler expansion file, the first line should be:

```
FILE_TYPE=EXPANSION_FILE;
```

This error should not occur if the design was compiled for LOGIC and the expansion file was not manually edited. Try compiling the design again before re-packaging.

ERROR #98: This property has already been specified

Either the FILTER_PROPERTY or the PASS_PROPERTY was specified more than once in the directives file. The Packager allows only one occurrence for each of these directives. Delete the extra directive(s).

ERROR #99: Error limit exceeded

The number of errors has exceeded the error limit set with the MAX_ERRORS directive. (The default value is 1000.) The Packager halts after printing this error message.

ERROR #100: Assertion check failure. Save PSTLOG file

An assertion check has been detected. Assertion errors indicate a problem internal to the Packager. If this error has not been caused by manual edits to a VALID file, save all input and output files and contact VALID.

ERROR #101: Compiler expansion file not found

The Compiler expansion file was not found. Make sure that the expansion file has been produced and that it resides in the current working directory.

ERROR #102: Compiler expansion file has wrong type

The Compiler expansion file does not have the correct file type for the Packager. The file type is found in the first line of the file. For a Compiler expansion file, the first line should be:

```
FILE_TYPE=EXPANSION_FILE;
```

This error should not have occurred if the design was compiled for LOGIC, and if the Compiler expansion file was not hand edited. Try compiling again (for LOGIC) before re-packaging.

ERROR #103: Library parts file has wrong type

A file defined by the directives as a library file has the wrong type. The first line of a library file should be:

```
FILE_TYPE=LIBRARY_PARTS;
```

To fix the error, make sure that the file is indeed a library file, and that its first line is as above.

ERROR #104: Library part has pin without PIN_NUMBER

A pin on a library part has no PIN_NUMBER. Add a PIN_NUMBER property to the pin on the library part.

ERROR #105: Incompatible number of sections for pin

The number of sections, as described by the pins of a library part, is inconsistent. The PIN_NUMBER property of a pin gives the physical pin number for that pin in each section. For example, a PIN_NUMBER of '(1,4,9,13)' for pin Y means that there are four sections in this package, each containing the said pin number for pin Y. Check the library to make sure that the pin descriptions are correct.

ERROR #106: Pin has vector pins of different widths

The library description of a part has two or more vectored pins with different widths. The vectored pins on a part in a library must all have the same width. This error indicates a library error; the library part must be changed.

ERROR #107: Cannot open directives file

The Packager failed to open the directives file (PACKAGER.CMD). Most likely, this file is missing from the current working directory, or the user does not have permission to access this file.

ERROR #108: Part type used not found in libraries

There is a part type in the design which is not defined by a library entry. (LS00, for example, is a part type.) All part types in a design must have a corresponding part definition in a library file unless the part has an AUTO_GEN property attached to it. Make sure the directives file includes all of the libraries which were used to make the design.

ERROR #109: Library part does not define used pin

A pin on a part is used in the design but is not defined in the library description for the part. All of the pins of a part used in a design must be defined in the library part, unless the part is to be AUTO_GENed by the Packager. The

Packager prints the part type and the pin name after the error message. Check the library and change the part description.

ERROR #110: Duplicate pin number for section part

A pin is defined twice in the part library description. The part type and the name of the pin are printed after the error message. The library description for this part must be changed.

ERROR #111: Unused.

ERROR #112: Run stopped because errors were detected

Fatal errors were found while packaging the design. The Packager stops processing after printing this message.

ERROR #113: Cannot open library file

Either the LIBRARY_FILE directive is missing, or the file specified by this directive does not exist. Make sure that the directives file contains a LIBRARY_FILE specification and that this shows the correct path name for the library.

ERROR #114: Common pins must have same pin name

The Packager has found some common pins with different pin names and is very confused. It expects all common pins within a part to have the same name and has printed the names of the pins in question. Please check your library.

ERROR #115: No library chips file has been specified

No library file was specified in the LIBRARY_FILE directive or the LIBRARY directive. The Packager needs the path name of the library to access physical information about the parts used in the design. To fix this error, give the library file name as follows:

```
LIBRARY_FILE 'library_file_name';
        or
LIBRARY 'short_library_name';
```

where library_file_name is the full path name of the file, and 'short_library_name' is a short name for one of the standard libraries of Valid known to the Packager. See the reference section on Packager directives for more details.

ERROR #116: Design using both GROUP and LOCATION_CLASS properties

The Packager detected both the GROUP property and the LOCATION_CLASS property in the same design. These two properties, though similar, cause the Packager to assign parts to packages differently, and they cannot be used together in the same design. Both are "body-properties" associated with logical parts.

The Packager guarantees to assign parts to packages according to these properties only if one (and not both) of the two properties is used in the design. Although the Packager proceeds with the assignment when both are present, it does not guarantee to meet all of the conditions for both properties. Delete all instances of either property from the design.

ERROR #117: Could not fit logical part at LOCATION

There are more logical parts with a LOCATION value than can be put into a physical part. For example, when an LS00 is given a SIZE value of 8 and a LOCATION value of U1, this will cause an error because an LS00 package has only 4 sections. This error can also occur if too many parts are forced into a package by feedback. If this error occurs, the LOCATION property should be removed or changed on some logical part, or the feedback file should be changed.

ERROR #118: Different part types have same LOCATION

The Packager found two logical parts of different part types with the same LOCATION value. An example of this is a LOCATION value of U1 for both an LS00 and an LS04. Since it is not possible to put two logical parts of different part types into one physical package, the LOCATION property value of one of the parts should be changed.

ERROR #119: Logical part matches no physical section

There is a discrepancy between a logical part in the design and its library description, or there is illegal use of a vectored pin. Every logical part in the design must have a library description unless the part is to be AUTO_GENed by the Packager. The part in error is printed following the error message and should be checked against the library representation.

ERROR #120: Wire-gate has multiple outputs

Sorry, no documentation for this error yet.

ERROR #121: Wire-gate has no output

Sorry, no documentation for this error yet.

ERROR #122: Inconsistent drive values

The Packager detected output values of pins on the same net which have different signs for one of the logic states. The Packager expects all outputs for either the High or Low logic state to have drive currents in the same direction. Consider the following example where a net has two output pins with these loading values:

```
Logic State                        LO   HI
Pin 1       OUTPUT_LOAD=(-8.0,-0.4)
Pin 2       OUTPUT_LOAD=( 8.0,-0.4)
```

The output values for these two pins have different signs for the low logic state and will cause a Packager error. To fix this error, check the library to make sure that all outputs on the same net have drive values of the same sign for each of the logic states. The IGNORE_0_LOADING or IGNORE_1_LOADING pin properties may also be used if load value checks are not desired.

ERROR #123: Zero drive output

The outputs have zero drive while there are input loads on the net. Check the library to make sure that the outputs have the correct drive values. Also make sure that the outputs on this net have enough loading capacity for the inputs connected to the net.

ERROR #124: Insufficient output drive

An output pin has insufficient drive for all the input(s) on its net(s). Although this pin may not cause a loading error for some particular nets that it is tied to, the output drive of this pin is too low for the total input loading that it must support. To fix this error, the user may use another part with higher output drive, or increase the TIMES property value of this component, or buffer the output to increase its drive.

ERROR #125: Inconsistent load values

Input load values have different signs. For either the high or low logic state, the Packager expects all inputs in that state to have the same current direction. To fix this error. check the library to make sure that all inputs on the same net have load values of the same sign. The user may also attach to the pin the IGNORE_0_LOADING or IGNORE_1_LOADING properties if load value checks are not desired (as in the case of connectors).

ERROR #126: All versions used

Sorry, no documentation yet.

ERROR #127: Inconsistent drive/load values

The input load values and output drive values on a net have the same sign; some input and output currents are in the same direction. Check the library to make sure the drive and load values have opposite signs. You may also attach to the pins the IGNORE_0_LOADING or IGNORE_1_LOADING properties if load value checks are not desired (as in the case of connectors).

ERROR #128: Outputs cannot drive their input loads

Output drive values are insufficient for the inputs on the net. The Packager compares the lowest drive value against the total input load values to insure that any of the outputs has sufficient drive for all of the input loads. This error is issued if the output drive is too low. To fix this error, you can use other output gates with higher output drive, or increase the TIMES property value of the output components, or buffer the outputs to increase their drive.

ERROR #129: Individual load too large for outputs

The output drive values are insufficient for the largest of the input loads on the net. For each net, the Packager compares the smallest output drive against the largest input load value. You can change an output gate to another with higher output drive, or increase the TIMES property value of the output component, or buffer the output to increase its drive.

ERROR #130: Input to wire-gate is also an output

An input and output of the same wire-gate are connected to the same net.

ERROR #131: No output on net

An input pin is not connected and hence gets no drive current. Make sure that the net printed by the Packager is wired as intended. Note that this message does not indicate an error if the pin is a primary input. In this case, you may want to attach the NO_IO_CHECK property to the pin to suppress input/ouput checks. This property has the form:

```
NO_IO_CHECK = logic_state
where logic_state may be 0, 1, or BOTH
```

If I/O checks are not desired for either logic state, use the value BOTH. To quiet this error message altogether, use the directive SUPPRESS 131.

ERROR #132: No input on net

An output pin has no subsequent blocks to drive. Make sure that the net printed by the Packager is wired as intended. Note that this message does not indicate an error if the pin is a primary output. In this case, you may want to attach the NO_IO_CHECK property to the pin to suppress input/output checks. This property has the form:

NO_IO_CHECK = logic_state
where logic_state may be 0, 1, or BOTH

If I/O checks are not desired for either logic state, use the value BOTH. To quiet this error message altogether, use the directive SUPPRESS 132.

ERROR #133: Illegal value for NO_LOAD_CHECK property

The NO_LOAD_CHECK property has an illegal value. The allowed values for this property are:

LOW
HIGH
BOTH
TRUE

The LOW and HIGH values inhibit load checking in the low and high state respectively. The BOTH and TRUE values inhibit load checking in both states. Any other values for this property cause an error.

These properties can be applied to pins, bodies, or nets. Note that attaching the NO_LOAD_CHECK property to a body suppresses load checking for every pin of the body; when attached to a net, this property suppresses load checking for the entire net; and when attached to a pin, load checking is suppressed ONLY for this pin, and not for other pins on the same net.

WARNING #134: Unused.output versions

The output current of some gate(s) exceeds that required for the inputs on the net. This is probably due to a TIMES property with too high a value. The Packager prints those output gates that are unnecessary to drive the inputs. To make the design leaner, reduce the TIMES value associated with the gates in question.

ERROR #135: Illegal value for WIRE_GATE property

The Packager finds a WIRE_GATE body property with an illegal value. The allowable values for this property are AND and OR. Any other WIRE_GATE values will cause error.

ERROR #136: Unused.

ERROR #137: Zero load values should not be used

The load value for a pin (found in library) is zero for either the high or low logic state. The Packager does not accept zero for a load value. If a pin presents no load on the net for a particular state, its load value should be specified as *. For example, a pin with no ouput load values for both states would have the following property:

```
OUTPUT_LOAD =(*,*)
```

ERROR #138: Illegal value for NO_IO_CHECK property

The NO_IO_CHECK pin property has an illegal value. The allowable values for this property are:

```
LOW
HIGH
BOTH
TRUE
```

The LOW and HIGH values inhibit input/output checks in the low and high state respectively. The BOTH and TRUE values inhibit I/O checks in both states. Any other value for this property causes an error. Note that NO_IO_CHECK is a pin property. When attached to a pin, it suppresses I/O checks ONLY for that pin, and not for other pins on the same net.

ERROR #139: Multiple outputs have incompatible type

Outputs of different types are tied together without explicit permission. Output pins may be tied together only if they are of the same output type, or if the ALLOW_CONNECT property has been added to the bodies. This error can be fixed by either adding the ALLOW_CONNECT property to the logical body on the drawing, or by adding the OUTPUT_TYPE property to each of the pins of the same type that are to be tied together. The OUTPUT_TYPE property values must match and be of the form:

(output type, logic function)

The standard OUTPUT_TYPE values for parts in Valid libraries are:

OC,AND	{ open collector; AND logic funtion }
OE,OR	{ open emitter; OR logic funtion }
TS,TS	{ TRI-STATE; logic function handled specially }

ERROR #140: Error detected during feedback

Errors were detected during the feedback process. Other error messages list the specific errors.

ERROR #141: Feedback file has wrong file type

The Packager opened one of the user-generated files during feedback, and it was the wrong type. A user-generated file is identified by its first line, which has the form:

FILE_TYPE = file type

The files that the Packager may use during feedback and their file types are as follows:

Files	File Type
Physical Part Designator Transformations (PSTPRTX)	PART_TRANS
Physical Net Name Transformations (PSTNETX)	NET_TRANS
Physical Section Transformations (PSTSECX)	SECTION_TRANS
Feedback Net List (PSTFNET)	FEEDBACK_NETLIST

Check the headers of the feedback files and make sure that their file types are correct.

ERROR #142: Old physical part does not exist

This error occurs during feedback while the Packager is attempting to swap sections using the physical section transformation file (PSTSECX). The Packager reads this file for the name of the old physical part, and tries to search for the part among those used in the design. In case of failure, it issues error 142 and prints the name of the physical part that it could not find. Check this feedback file to make sure that all physical parts referenced are present in the design.

ERROR #143: Pin number does not define section

This error occurs during feedback while the Packager is attempting to swap sections using the physical section transformation file. The Packager reads the file for the physical part and the pin number of the old section, and it tries to locate a section on the physical part from this information. It generates error 143 if no section is uniquely defined by the pin number. Check the file PSTSECX to make sure that the pin number exists for the physical part, and that this pin is not a common pin or a power pin.

ERROR #144: Physical net does not exist

The Packager cannot find the physical net name given in the Feedback Net List file (PSTFNET). Each physical net name in this file must correspond to an existing net in the design, i.e., the feedback net name must match exactly one of the net names in the Expanded Net List (PSTXNET).

This error could be caused by manual edits to the Feedback Net List file which cause the misspelling of a name. Otherwise, the error may be caused by changing the net name from the last run to the current run (i.e., by changing the design without recompiling and repackaging with USE_STATE_FILES on, or by performing Feedback Net Transformation, or by changes to the NET_NAME_LENGTH directive).

Remember that to do feedback, the design must remain exactly the same from the last Packager run to the current feedback run. If any of the above changes have occurred, back up and restore the design to the old state. To find the previous name of the net that is now not recognized by the Packager, look in the Expanded Net List (PSTXNET) for a list of all nets known to the Packager.

ERROR #145: Part type not present in design

A part type in the feedback net list file (PSTFNET) was not found in the design. The feedback net list file specifies the net transformations that are to take place. The information in this file includes the name of the signal to be reconfigured and the name of the new physical part and the corresponding part type to which the signal will be connected. This part type should already be in the design; the Packager issues an error message otherwise. Check the feedback net list file to make sure that the specified physical parts and part types exist in the present design. Remember that the design is not to be modified before packaging with feedback.

ERROR #146: Old section not in use

This error occurs during feedback while the Packager is attempting to swap sections using the physical section transformation file (PSTSECX). The Packager reads the file for the physical part and the pin number of the old section. It tries to access the corresponding logical part, and then performs the section transformation on this logical part. Error 146 appears if the Packager cannot find a logical component in the design that corresponds to the information from the feedback file. To fix this error, check the section transformation file to make sure that the correct physical part name and pin number are given. The physical part name and pin number of logical parts can be found in the cross reference file (PSTXREF) or by doing back annotation.

WARNING #147: Feedback section matches multiple parts in design

The Packager finds more than one logical part matching a section specified in the feedback netlist file (PSTFNET). While doing feedback, the Packager tries to match the description of a section in the feedback file with logical parts in the design. It prints a warning if it cannot find a unique match, but proceeds to assign the feedback part to one of the matching parts in the design. A possible cause for this error is feedback of parts of the same type that are connected in parallel. Consider a design having a number of resistors in parallel. The information from the Feedback Netlist file, namely the part type and the nets connected to the part, is insufficient for the Packager to isolate a single resistor.

When this error occurs, check the feedback net list file to make sure that the information in this file is correct and uniquely identifies the desired part to the Packager. If you have further difficulty swapping components, try doing feedback with the physical section transformations file (PSTSECX).

ERROR #148: Pin in feedback net list not on part

A pin specified by the user in the feedback net list file (PSTFNET) is not on the physical part. When the Packager encounters an instance of a pin in the feedback net list file, it tries to match this pin with one of the pins on the physical part in the library. If there is no match, error #148 is issued. Make sure that the library describes all of the pins for the physical part, and that only defined pins are referenced.

ERROR #149: Match not found for feedback section

This error is generated when the Packager cannot find the section matching that described by the nodes in the feedback net list file (PSTFNET).

To get rid of this error, check the feedback net list file to make sure that:

1. An entry is listed for each of the pins of the section. (A missing pin is assumed to be connected to the net 'NC'.)
2. The part_type is spelled correctly. (Make sure the EXACT part_type name is given. The Packager distinguishes between 'LS00', for instance, and '74LS00'.)
3. Each net name for the section must match exactly an existing net name in the design. Make sure that the directive NET_NAME_LENGTH has not been added, deleted, or changed from the last Packager run.

Also be sure that:

1. The design has not been changed without recompiling and repackaging.
2. The library used in the current Packager run is the same as the one used during the previous run.

Following the message, the Packager prints the feedback entries that it is trying to match with a section from the design and the partial matches it found for each entry. Make sure that these entries are as intended for the specific section. If the Packager prints redundant entries, it means some pins are missing for this section, and the Packager fills these in as pins connected to 'NC' nets. If some partial matches are not generated, try the "verbose" option of the FEEDBACK_ORDER directive for the complete list of partial matches. This option is:

```
FEEDBACK_ORDER -V FEEDBACK_NETLIST;
```

ERROR #150: Feedback file may be used only once

The Packager encountered more than one instance of a feedback file type given in the directive

```
FEEDBACK_ORDER file_type
```

The feedback file types that you can specify with this directive include:

```
PART_TRANS
SECTION_TRANS
NET_TRANS
FEEDBACK_NETLIST
```

You can enter more othan one file type, but you cannot enter a given file type more than once.

ERROR #151: Illegal value for feedback file name

The Packager encountered an unknown feedback file type in the directive

```
FEEDBACK_ORDER file_type
```

The feedback file types that you can specify with this directive include:

PART_TRANS
SECTION_TRANS
NET_TRANS
FEEDBACK_NETLIST

Any other file types will cause an error.

ERROR #152: FEEDBACK_ORDER directive allowed once

The FEEDBACK_ORDER directive was given more than once in the directives file (PACKAGER.CMD). Delete the extra directive(s).

ERROR #153: Section transformations file not found

The Packager was instructed to swap sections using the physical section transformations file, but cannot find the file. When using the directive FEEDBACK_ORDER SECTION_TRANS, you must supply the physical section transformations file (PSTSECX) in the current working directory. This file shows the pairing of old physical section to new physical section. See the Packager documentation on feedback processing for a complete description of the format of this file.

ERROR #154: Part transformations file not found

The Packager was instructed to swap physical parts, but cannot find the physical part designator transformation file. When using the directive FEEDBACK_ORDER PART_TRANS, you must supply the physical part designator transformations file (PSTPRTX) in the current working directory. This file shows the pairing of old physical part designator to new physical part designator. See the Packager documentation on feedback processing for a complete description of the format of this file.

ERROR #155: Feedback net list file not found

The Packager was instructed to perform changes in the design using the feedback net list file, but can not find the file. When using the directive FEEDBACK_ORDER FEEDBACK_NETLIST, you must supply the feedback netlist file (PSTFNET) in the current working directory. See the Packager documentation on feedback processing for a complete description of the format of this file.

ERROR #156: Net transformations file not found

The Packager was instructed to swap nets using the physical net name transformations file, but cannot find the file. When using the directive FEEDBACK_ORDER NET_TRANS, you must supply the file PSTNETX in the current working directory. This file contains a list of the pairs of old physical net name and new physical net name. See the Packager documentation on feedback processing for a complete description of the format of this file.

ERROR #157: Attempting to do feedback on wrong design

The ROOT_DRAWING name is not given in the feedback file header, or the ROOT_DRAWING name does not match the current design name. Make sure that you are using the right Compiler expansion file and that the root drawing name is given correctly in the feedback file.

ERROR #158: Unknown user assigned section number

An SEC (section) value assigned to a pin is out of the range known to the Packager. For example, it is an error to assign an SEC value greater than the number of sections in a physical part.

ERROR #159: SIZE/TIMES <> 1 for section assignment

Sorry, no documentation for this error yet.

ERROR #160: SEC not the same on all pins of section

Different SEC (section) values are assigned to the pins of a logical part in the design. To assign a logical component to a physical section, you should apply the SEC property to either the body or one of the pins in a section. This error occurs when the SEC properties attached to pins of the same section are not of the same value. Change this in the design.

ERROR #161: SEC same on some of the sections of logical part

The SEC (section) property was used illegally on some HAS_FIXED_SIZE part. A HAS_FIXED_SIZE part is a logical part that contains more than one section. All of the sections within the same logical part must go to the same physical package, and thus cannot get the same SEC value. Make sure that all of the sections of this component have unique SEC values.

ERROR #162: Logical part has already been allocated

A logical part was allocated to a physical part before the Packager assigned it. This error is internal to the Packager and should be reported to Valid.

ERROR #163: Could not find pin number on part type

The pin number specified in the pin swap file does not exist in the definition of the part type. The Packager prints the pin number and the path name of the logical part following the error message. Check the library definition for this part.

ERROR #164: Pins are not swappable

The Packager cannot swap the pins as specified by the user. To be swappable, pins must be in the same section and must have the same PIN_GROUP value. Make sure that the pins to be swapped are logically equivalent and are in the same section of the part, and then change the library so that these pins have the same PIN_GROUP value. If PIN_GROUP checking is not desired for the entire design, use the directive USE_PIN_GROUP OFF.

ERROR #165: Value out of range

The Packager was reading the pin swap file and found a value out of the allowed range. The offending number could be the size offset of a pin, or the version number of the part, or the bit subscript. The Packager prints the value in question following the error message.

ERROR #166: Can't change user defined pin assignment

The Packager cannot perform pinswapping because of previous user assignments.

ERROR #167: Pin Swap file has wrong file type

The Packager opened the pin swap file and the FILE_TYPE was the wrong value.

ERROR #168: Unused.

ERROR #169: Unused.

ERROR #170: Illegal value for BODY_TYPE property

The Packager was reading the library CHIPs file, and a part has the BODY_TYPE property with an illegal value. The two allowed values for the BODY_TYPE property are:

FLAG_BODY
STL_CHIP_DEF

Any other values cause an error.

ERROR #171: LOCATION in multiple GROUPs

The LOCATION property is forcing logical parts of different GROUPs into the same physical part. The GROUP property requires that logical parts of different GROUPs be assigned to different physical parts. It is an error for such parts to have the same LOCATION property value.

WARNING #172: Directive has already been specified

The directives file contains more than one occurrence of the HEADER_FILE directive. Delete the extra directive.

ERROR #173: Unused.

ERROR #174: Cannot open temporary bindings file

The Packager failed to open a file for storing new logical to physical bindings while doing feedback. This is usually due to some protection or disc space problem. Make sure that the user has write permission in the current working directory and that there is enough space to run the Packager.

ERROR #175: LOCATION in multiple LOCATION_CLASSes

Logical parts in different LOCATION_CLASSes cannot be assigned to the same physical part and hence cannot have the same LOCATION property value.

ERROR #176: Cannot alloc part due to LOCATION_CLASS

It is an error to feed back logical parts of different LOCATION_CLASSes to the same physical part.

ERROR #177: Part bindings file has wrong type

The Packager opened the Part Bindings State File (PSTSTAT) and it was the wrong type. State files are generated and maintained only by the Packager. You should NOT change the state files in any way. Delete all of the state files (PSTSTAT, PSTPRTB, PSTSIGB, PSTPSWP) and rerun the Packager to regenerate these files before proceeding.

ERROR #178: Signal bindings file has wrong type

The Packager opened the signal bindings state file (PSTSIGB) and it was the wrong type. State files are generated and maintained only by the Packager. You should NOT change the state files in any way. Delete all of the state files (PSTSTAT, PSTPRTB, PSTSIGB, PSTPSWP) and rerun the Packager to regenerate these files before proceeding.

ERROR #179: Physical net name reused in state file

A physical net name occurs more than once in the Signal Bindings State File or the Expanded Net List File. The name of the offending net is printed after the error message. This is most probably caused by the illegal assignment of a net name to more than one net. To fix this error, either rename the offending nets using feedback, or back up to the previous stage by deleting the state files (PSTSTAT, PSTPRTB, PSTSIGB, PSTPSWP), and then rerunning the Packager with the directive USE_STATE_FILES ON.

ERROR #180: Error detected in part bindings file

An error occurred while the Packager was in the process of assigning physical sections. See other error messages for details on the nature of the problem.

ERROR #181: Error detected in signal bindings file

Sorry, no documentation on this error yet.

ERROR #182: Cannot open file for write

The Packager failed to open an output file. This is usually due to some protection or disc space problem. The name of the file is printed. Check the protections for the file and the user to make sure they are compatible.

ERROR #183: Cannot close output file

This error most likely occurs because the file has not been opened or because the file is already closed. Make sure that the file name printed with this error message is open and has not been previously closed.

ERROR #184: Cannot close input file

The file has not been opened or has already been closed. Make sure that the file name printed with this error message is open.

ERROR #185: LOCATION too long

The LOCATION property value assigned to the part is longer than allowed. The limit on the length of a physical part name is set by either the PART_NAME_LENGTH directive or a default value of 16, whichever is lower. To correct this error, either shorten the value of the LOCATION property or change the PART_NAME_LENGTH directive to allow for a longer name.

ERROR #186: Error detected in state file

The Packager opened the design information state file (PSTSTAT) and found that it was of the wrong type. State files are generated and maintained only by the Packager. You should NOT change the state files in any way. Delete all of the state files (PSTSTAT, PSTPRTB, PSTSIGB, PSTPSWP) and rerun the Packager to regenerate these files before proceeding.

ERROR #187: Cannot do feedback due to changes/errors

This error can be caused by:

1. Absence of the state files (e.g., the last Packager run had the directive USE_STATE_FILES OFF, or the state files were deleted.

2. Manual edits to the state files or to the Compiler expansion file.

3. Changes to the design without a subsequent recompile and repackage to generate current state files before the feedback attempt.

State files must be present if feedback is to take place. If there are no state files, run the Packager once with the directive USE_STATE_FILES ON. State files are generated and maintained only by the Packager. If you have modified the state files, delete them and run the Packager again before doing feedback. Also, the Packager expects the design to remain the same from the last run to the current run if there is feedback processing.

ERROR #188: Cannot change LOCATION property value

The Packager cannot overwrite a user-assigned LOCATION by feedback unless you use the directive HARD_LOC_SEC OFF. To fix this error, either delete the LOCATION property of the part from the design and recompile, or change the feedback file to avoid conflict. You can also enter HARD_LOC_SEC OFF into the directives file for the Packager. This tells the Packager that feedback assignments may overwrite previous LOCATION and SECTION assignments. (Be careful! Be sure that you want to overwrite the current LOCATION/SEC properties.) If you use HARD_LOC_SEC OFF, be sure to back annotate and recompile to make the changes permanent. Then reset HARD_LOC_SEC back to ON to protect your new assignments.

ERROR #189: Cannot change section assignment

The Packager was in the process of swapping sections and found that the logical part to be swapped was already assigned to a physical section. Unless the user has been tampering with the state files, this error is internal to the Packager and is not caused by the user. The problem should be reported to Valid.

ERROR #190: Pin found that is neither IN nor OUT

Sorry, no documentation for this error yet.

ERROR #191: Unused.

ERROR #192: Unused.

ERROR #193: Signal name has length zero

A user-assigned signal name is empty, or it contains only the character '-'. Make sure that every signal name contains at least one character.

ERROR #194: Invalid margin values

The margins of the output are set so large that there is no room left for printing. This is an error internal to the Packager and should be reported to Valid.

ERROR #195: Synonym file has wrong type

The Packager opened the Compiler synonym file and found it was the wrong type. This is probably caused by user tampering with the file header. Try compiling the design again before running the Packager.

ERROR #196: Drawing name too long

The drawing name that the Packager read from the Compiler synonyms file was too long. Shorten the name of the design drawing.

ERROR #197: Expected DRAWING

The Packager cannot find the DRAWING entry in the Compiler synonyms file. This may be due to user tampering with this file. Try compiling the design again before rerunning the Packager. If the error still occurs, then the problem is probably within the Compiler.

ERROR #198: Not all outputs have same OUTPUT_TYPE

OUTPUT_TYPE is a pin property on output pins. Output pins on the same net must have the same OUTPUT_TYPE unless the ALLOW_CONNECT property is present.

ERROR #199: Cannot determine WIRE_TYPE of the net

Sorry, no documentation for this error yet.

ERROR #200: Cannot open CMPSYN for net annotation

The Packager failed to open the Compiler Synonym file for output of synonym signals to the back annotation file (PSTBACK). This is usually due to some protection or disc space problem. Check the protections for the file and the user to make sure they are compatible.

ERROR #201: Overlapping bit subscripts for pin

The Packager encountered confusing information about a pin. The part type for this pin is printed in the error message. This error could be caused either by incorrect modeling of the parts in the library or by illegal use of signal names in the design.

A common violation is to specify a part type more than once in the library. Check the CHIPS file to make sure that the part type printed is uniquely described. If there is more than one library, be sure that the part type is not multiply defined in different libraries.

Another possible cause for this error is the breaking of a vector pin into subranges. Although each bit of a vector pin may be used in the design, subranges are not allowed. For example, a vector pin A<7..0> may be specified as A<7>, A<6>,..., A<0>. However, it is illegal to represent this vector pin as A<7..4> and A<3..0> in the drawing.

ERROR #202: Net name in bindings file is too long

The Packager was reading the signal bindings state file and found a physical net name longer than it expected. This usually happens when the user changes the net name before running the Packager with the directive USE_STATE_FILES ON. Delete the state files (PSTSTAT, PSTPRTB, PSTSIGB, PSTPSWP) and rerun the Packager with the state files directive before proceeding.

ERROR #203: Cannot create unique physical net name

The Packager ran out of unique names for physical nets. The maximum length set for physical net names was too short for the number of nets in the design. Increase the operand for the directive NET_NAME_LENGTH in the directives file.

ERROR #204: PHYS_DES_PREFIX value is too long

The PHYS_DES_PREFIX property value that you assigned to a body is longer than the Packager allows. The Packager prints the path name of the logical part in question.

ERROR #205: Cannot create unique physical part name

The Packager ran out of unique names for physical parts. The maximum length set for physical part names was too short for the number of parts in the design. Increase the operand for the directive PART_NAME_LENGTH in the directives file.

ERROR #206: HAS_FIXED_SIZE part found in the library

A part with the HAS_FIXED_SIZE property was found in the part library. The Packager expects all parts to be SIZE-expandable, and does not understand that a HAS_FIXED_SIZE part in the library can also have the SIZE property. To be able to use a HAS_FIXED_SIZE part in a design, you should model only a single-size version of the part in the library. Remove this property from the library part.

ERROR #207: HAS_FIXED_SIZE value does not match part

The HAS_FIXED_SIZE property value of a logical part was incorrect. A quadruple MUX LS157, for example, is logically equivalent to four single LS157's, and should have a HAS_FIXED_SIZE property value of four. Any other value is an error. This error is probably caused by incorrect library information.

ERROR #208: Could not read Expanded Net List

The Packager cannot open the Expanded Net List (PSTXNET) while assigning net names to signals in the design. Make sure the user has permission to read this file.

ERROR #209: Cannot do feedback with state files off

The Packager was instructed to do feedback with the state files off. When the Packager sees the directive FEEDBACK_ORDER, it expects the state files to be used. To fix this error, first get the state files by running the Packager with the directive USE_STATE_FILES ON and no FEEDBACK_ORDER. Then add the FEEDBACK_ORDER directive again for packaging with feedback.

ERROR #210: Unused.

ERROR #211: Unused.

ERROR #212: Unused.

ERROR #213: Unused.

ERROR #214: Unused.

ERROR #215: Unused.

ERROR #216: Unused.

ERROR #217: Unused.

ERROR #218: Unused.

ERROR #219: Unused.

ERROR #220: Cannot open physical part table file

The Packager failed to open the physical part table file. This is usually due to some protection or disc space problem. Check the protections for the file and the user to make sure they are compatible.

ERROR #221: Expected PART

The keyword PART is missing at the beginning of a part type table. The first line in the part type table should have the form:

PART 'part name'

where PART NAME is the name of the part type to be redefined by the information in the table. An example is:

PART '1/4W RES'

ERROR #222: Unexpected characters at end of line

There are redundant characters at the end of a line for a part type property entry. While processing the physical part table, the Packager expects each part type property to be specified on a single line in the format:

property name = property value

To fix this error, remove any extra characters following the part type property.

ERROR #223: Expected END_PART

The key word END_PART is missing at the end of a part type table. Insert the END_PART after the part type table.

ERROR #224: Added property not unique on part type

A property was specified more than once for a part type in the physical part table. The Packager prints the property name and the part type following the error message.

ERROR #225: Unknown property attribute

An unknown property attribute was specified in the physical part table. Currently, the only attribute understood by the Packager is the OPT attribute, which indicates whether a property is optional on an instance of a part. Any other property attributes in the physical part table will cause error. Please see the Packager chapter section on physical part tables for more details.

ERROR #226: Property specified more than once

A property was specified more than once in the physical part table. The Packager prints the name of the property following the error message.

ERROR #227: Duplicate physical part table entries

A physical part table entry was specified more than once in the physical part table. The Packager prints the entry after the error message.

ERROR #228: Property not found on logical part

An instance property in the physical part table was missing its value. You may add or modify the values of instance-specific properties on logical parts by giving new property values in the physical part table. If such a property is not optional (as indicated by the attribute OPT following the property name), the Packager expects to find a value given for each instance property. Make sure that the appropriate values are given in the table for non-optional properties.

ERROR #229: Physical part table entry not found

The Packager cannot find a logical part with the instance properties specified in the Physical Part Table. Check the physical part table to make sure that the entry the Packager printed has the correct properties.

INDEX