CHAPTER 8

PACKAGER

Overview of the Packager

8.1 INTRODUCTION

The Packager provides the interface between the SCALDsystem and the user's physical design environment. The files output by this program are intended to supply all the information needed by physical design systems. The Packager reads the expansion file created by the Compiler and produces files for the user's physical design system. These output files include net lists, part lists, etc.

State files can be generated to maintain a consistent description of the design from one Packager run to another. The Packager can also read files created by the user's physical design system that specify changes in the physical design or additions to it that need to be reflected in the output files. These changes are then made to the state files during the next execution of the Packager.

8.2 A SUMMARY OF THE PACKAGER'S FUNCTIONS

The Packager performs several functions. These are summarized below with a short description. A complete description of each will be presented later in this section.

1. Expand parts with SIZE properties.

   Parts with SIZE properties need to be expanded into (possibly) several parts. The signals connected to the original part are assigned to the expanded parts.

2. Create versions of parts that have TIMES property.

   New versions of those parts that have TIMES properties are created. The signal versions (the outputs of versioned parts) are allocated as needed in the rest of the design.

3. Remove phantom WIRE-OR and WIRE-AND bodies.

   All WIRE-OR and WIRE-AND bodies used in the design are removed and replaced with explicit wire tying. New versions of signals are used to implement wire

tying (see the section on WIRE-ORs and WIRE-ANDs).

4. Logical to physical mapping.

   Each logical part, signal, and pin is given a physical name. These names are originally created by the Packager but can be changed by the user.

5. Loading calculations.

   The Packager checks each net to make sure that the loading specifications of the parts are not violated.

6. Input and output checks.

   The Packager checks that there is at least one input and one output pin connected to each net. This detects the cases where there are dangling inputs or outputs.

7. Output file creation.

   The Packager creates several files containing the results of the run. These files include a net list, parts list, logical design change summary, etc.

8. Feedback of physical design changes.

   The physical design created by the Packager may differ from the design created by the physical design system. All of these changes can be fed back into the Packager to modify its understanding of the actual physical design.

The Packager also provides preliminary support for engineering changes. The treatment of ECOs in general will be addressed at a later date; the current Packager is not intended to be a solution to those problems.

8.3 PACKAGER INPUT FILES

The following files are read by the Packager. Some of them are created by the Packager, some are created by the Compiler, and some are created by the designer (or the designer's physical design system). The specific formats of these files is described later in this section.

1. Packager Directives File

   This file contains commands used to direct the execution of the Packager and is created by the designer with a text editor.

2. Compiler Expansion File

   This is the output file from the Compiler and contains a body ordered description of the design. All user defined properties and signals are found here.

3. CHIPS File

   This file contains descriptions of the physical part types used in the design. Each entry contains the physical part type, pin numbers, section descriptions, default properties, current loading values for both input and output pins, and family specification. This file is generated by the Compiler from a special set of library drawings.

4. Physical Part Tables

   These files contains tables describing how new part types can be created from existing part definitions on an instance by instance basis. This is useful for discrete parts such as resistors and capacitors. These files are created by the designer with a text editor.

5. Logical Signal Name to Physical Net Name Bindings

   This file contains bindings of logical signal names to physical net names. Each logical signal name is in cannonical form. It also contains a path name specifying the signal instance. The physical net name is a physical name string. Each signal name pair represents a single bit. This file is maintained by the Packager and is not to be written or changed in anyway by the user.

6. Physical Net Name Transformations

   This file is used to change the physical net names assigned by the Packager. It consists of a list of

old physical net names and new physical net names. It is created by the user or the user's physical design system.

7. Logical to Physical Part Designator Bindings

This file contains a list of logical to physical designator bindings. Each entry consists of a logical part designator and its corresponding physical part designator. Each entry also contains a section specification which binds logical pin designators to physical pin designators. This file is maintained by the Packager and is not to be written or changed in any way by the user.

8. Physical Part Designator Transformations

This file contains changes of old physical part designators (the ones assigned by the Packager during the last run of the program) to new physical part designators (as assigned, possibly, by some layout program). This file provides a means by which the physical part designators assigned by the Packager can be changed. This file is generated by the user or the user's physical design system.

9. Physical Section Transformations

This file contains changes of old physical sections (the ones assigned by the Packager during the last run of the program) to new physical sections (as assigned, possibly, by some layout program). This file provides a means by which physical sections assigned by the Packager can be changed. This file is generated by the user or the user's physical design system.

10. Feedback Net List

This file contains a net list sorted by physical part designator of the current physical design. The only restriction is that the physical net names must not have changed or the new physical net names have already been fedback before performing a net list feedback. The Packager will detect and perform all physical part designators changes, physical section swapping, and pin swapping from the feedback net list. The file is generated by the user or the

user's physical design system.

11. Signal Synonyms File

    This file is produced by the Compiler and lists all aliases for each signal that has a synonym in the design.

12. State file

    This contains information about the last run of the Packager. It is used to ensure that the Packager produces the same output for the same inputs.

13. Pin Swap File

    This file contains the pin swap information generated from a previous run of the Packager. This file is maintained by the Packager and is not to be written or changed in any way by the user.

FILES WRITTEN BY THE PACKAGER

The following files are created by the Packager. Some of them may be selectively disabled should the file not be needed. Some of these files are read by the Packager in subsequent runs of the program.

1. Execution Summary

   A summary of the Packager execution showing execution times, statistics, error messages, etc.

2. Cross References

   This file contains several cross references intended to provide information about the design that is not readily available in the drawings. These cross references include both physical and logical information and how they correspond. They are organized to provide access to the design as a whole or to a particular drawing.

3. Logical Signal Name to Physical Net Name Bindings

   This file contains bindings of logical signal names to physical net names. Each logical signal name is in SCALD format. It also contains a path name specifying the signal instance. The physical net name is a physical name string. Each signal name pair represents a single bit. This file is maintained by the Packager and is not to be written or changed in any way by the user.

4. Logical to Physical Part Designator Bindings

   This file contains a list of logical to physical designator bindings. Each entry consists of a logical part designator and its corresponding physical part designator. Each entry also contains a section specification which binds logical pin designators to physical pin designators. This file is maintained by the Packager and is not to be written or changed in any way by the user.

5. Expanded Net List

   A net list consisting of the physical net name, properties of that net, physical part type, physical part designator, physical pin name, and properties of the logical pins. Properties from the chips file are not output.

6. Expanded Parts List

   A part list consisting of physical part designators, logical to physical bindings, and the body properties of the logical parts. Properties from the chips file are not output.

7. Logical Changes Summary

   A list of the changes in the logical design from the last run of the Packager. Any changes in logical part to physical part assignments are listed to this file.

8. State File

   This contains information about the last run of the

Packager. It is used to ensure that the Packager produces the same output for the same inputs.

9. Pin Swap File

This contains the pin swap information generated by the Packager during a feedback net list. This file is maintained by the Packager and is not to be written or changed in any way by the user.

10. Log File

This file contains assertion check error messages, run time statistics, internal debug information, etc. used by Valid personnel when searching for information about program bugs. The user does not normally need to look at this file.

11. Back Annotation File

This file contains information added to the design by the Packager or the physical design system which the user may wish to see reflected in the drawings. The Graphics Editor reads this file and adds this information to the user's drawings.

12. Reports

This file contains several user reports generated by the Packager. These reports includes a part summary of the design and a list of spare physical sections.

13. New CHIPS file

This file contains the new part definitions that were created from processing physical part tables. This file can be used by other programs such as DIAL to differentiate the newly created part types.

Packager Reference Manual

8.4 INTRODUCTION

The following sections describe the various functions of the Packager.

8.5 SIZE EXPANSION

The SIZE property is used to generate a multiple-bit component, and connect it to a group of signals. The Packager generates SIZE number of logical parts and assigns a new logical part 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 specifies the width of the pin it is attached to. The Packager allocates this number of bits from the original signal to each logical part. If the original signal is not wide enough for all logical parts, the Packager begins allocating from the beginning of the signal again. See the documentation describing the construction of physical libraries for more detail.

The assignment of bits of a signal to physical parts is done sequentually so that adjacent bits are assigned to the same physical package if possible.

8.6 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 for the product 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 of the net 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.

Any part may have multiple versions of its outputs generated by attaching the TIMES property to the instance of the part. The number of versions of each output that will be generated is the value of the TIMES property. 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, and the Packager will divide the loads among the versions of an output so that the specified loading rules will be obeyed. In the process, it will generate 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 connects to, the Packager will attempt to divide the loads evenly among all output versions. If loading rules require more output versions than are specified, the excess number required will be determined and flagged as an 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.

Pins with the BIDIRECTIONAL property (see the section describing current loading) are considered to be output pins when versions of the part are created because of a TIMES property.

8.7 WIRE-GATE AND WIRE-TIE EXPANSION

In SCALD III, outputs of appropriate technologies may be connected together in two different ways:

- An explicit wire-tie.
- A wire-gate.

Wire-ties are simply the connection of two or more outputs to the same signal and are used primarily for connection of several drivers to a common bus. The signal present on the outputs is exactly that signal which is present on the bus, since the outputs are connected to the bus.

Wire-gates are used whenever signals are wired together to form the logical-or or logical-and of the signals (depending on the technology), and the signals are also used elsewhere in the design. In these cases, simply connecting to the same net (wire-tying) the outputs driving each signal is not acceptable, since it results in all the signals assuming the value of the logical-or or logical-and. The correct way to wire these signals together is to wire-gate the signals by connecting versions of the outputs which are used nowhere else in the design. By doing this, 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.

TIMES properties on wire-gates function exactly the same as on physical parts. Several versions of a net are generated, each with different output versions. For each version of the wire-gate output net, one version of each of the wire-gate input nets is used. If a net connects to a wire-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 wire-gate. If a wire-gate has no TIMES property, it is ignored when determining the number of versions of the output net, and assumes the TIMES value required by the net. Each version of the output net will use one version of each input net, whether the wire-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 wire-gate inputs.

After a number of versions of a net have been generated (by use of the TIMES property), the number of versions required by wire-gate inputs is first used. 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 wire-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 outputs driving the remaining inputs to the wire-and are not used. The same is true for wire-or's, with '0' and '1' reversed.

8.8 NET CHECKS PERFORMED BY THE PACKAGER

A net is a single bit signal and the nodes (parts and the specific pin) that are connected to it. There are several consistency checks performed for nets. These are:

1. Make sure every net is connected to at least one input pin and at least one output pin.

2. If a net is connected to more than one output pin, make sure all those pins have the proper technology (OC, OE, TS, etc.).

3. Make sure loading rules are not violated.

Each of the checks above is discussed below. Loading rules will be discussed in the next section.

INPUT AND OUTPUT PIN CHECKS

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. The presence of the OUTPUT_LOAD property on a pin indicates that the pin is an output and the presence of the BIDIRECTIONAL property indicates that the pin is both an input and 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 appear on an instance by instance basis in 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, NO_IO_CHECK applies to all pins of the body. When used as a pin property, NO_IO_CHECK applies only to the pin to which it is attached.

MULTIPLE OUTPUT CHECKS

Outputs can only be tied together if they are given explicit permission to do so. Permission is given by the OUTPUT_TYPE property attached to the output pins. The OUTPUT_TYPE property serves three purposes. First, it gives permission to the pin to be connected to other outputs. Second, it specifies the type of output so that only compatible outputs may be connected together. Third, it specifies the logic function created by tying the outputs together. When the Packager detects outputs tied together that do not have the OUTPUT_TYPE property or outputs tied together that hav incompatible OUTPUT_TYPE properties, it produces an error message indicating the output pins as well as the net name.

Each output pin that can be connected to other output pins must have the OUTPUT_TYPE property. The property value specifies the pin type and also the logic function created by tying the outputs together. The form of the OUTPUT_TYPE property value is:

(<output type> , <logic function>)

where <output type> is the output type name and can be any identifier (string of letters, digits, or '_' starting with a letter). The <logic function> is optional and specifies the logic function of outputs tied together and may be either AND, OR, or TS. See the Timing Verifier documentation for a description of how multiple output nets are simulated in the presence and absence of the logic function portion of the OUTPUT_TYPE property.

Outputs can be connected together only if they have the OUTPUT_TYPE property and the property values are the same. The parts in the 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; logic function handled specially }
```

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

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

The ALLOW_CONNECT property may appear on a library part (as in the case of a standard connector) or can appear on an instance by instance basis in the drawings.

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, ALLOW_CONNECT 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.

8.9 DEVICE LOADING CALCULATIONS

Once a design has been expanded into physical components, and the interconnection between them is 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). <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 (of the form n.m) that specifies the load in some consistent units.

The values used for some Valid libraries (such as the LSTTL library) is 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. Theses loading values are specified in mA (Amps x 0.001) and 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 has 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 (as, for instance, in the case of 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. There are two loading calculations performed: "0 state" loading and "1 state" loading. The "0 state" loading calculation proceeds as follows:

1. Find the minimum "0 state" value found in all the OUTPUT_LOAD properties of pins connected to the net.

2. Add up the "0 state" value found in all the INPUT_LOAD properties of pins connected to the net.

3. Calcuate the net loading by adding the minimum OUTPUT_LOAD to the INPUT_LOAD totals.

4. If the result has a different sign than the sign of the OUTPUT_LOAD value, report a loading error for the net.

For example, consider a net with four nodes (pins) with the following loading properties:

```
OUTPUT_LOAD = (3.0,-1.8)
INPUT_LOAD  = (-1.2, 0.2)
INPUT_LOAD  = (-1.2, 0.2)
INPUT_LOAD  = (-1.2, 0.2)
```

The "0 state" net loading value is -0.6. Since this value has a different sign than the value given in the OUTPUT_LOAD (3.0), the net violates loading rules; there are too many inputs for the given output drive capability.

The calculation of the "1 state" loading proceeds similarly. In the example used above, the "1 state" loading for the net would be -1.0 which is not an error (the sign of the value for the entire net is the same as the value for the output).

If a net loading error exists and may 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. This may, in turn, cause errors if not enough versions of the outputs exist. If a net has a loading error which cannot be fixed by more versions (such as 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 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".

SUPPRESSION OF DEVICE LOADING CALCULATIONS

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.

The NO_LOAD_CHECK property 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 may appear on a library part (as in the case of a standard connector) or can appear on an instance by instance basis in the drawings.

If the NO_LOAD_CHECK is used as a net property, it applies to all the pins on the net. When used as a body property, NO_LOAD_CHECK applies to all pins of the body. When used as a pin property, NO_LOAD_CHECK applies only to the pin to which it is attached.

SPECIFICATION OF UNKNOWN LOADING

Occasionally there are parts in a design that have pins with unspecified or unknown 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 is used to inform the Packager that loading is unknown and to suppress loading and I/O checks on the entire net if it appears on any pin of the net.

The UNKNOWN_LOADING property may appear in on a library part (as in the case of a standard connector) or can be used on an instance by instance basis in the drawings.

If the UNKNOWN_LOADING is used as a body property, it applies to all the pins of the body. When used as a pin property, UNKNOWN_LOADING applies only to the pin to which it is attached.

If one attaches the NO_LOAD_CHECK to a pin with either UNKNOWN_LOADING on 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 the NO_LOAD_CHECK. Likewise, attaching the NO_IO_CHECK to a pin will only suppress I/O checking only for the pin as specified by the value of the NO_IO_CHECK. This mechanism allows the user to "suppress" the effects of the UNKNOWN_LOADING property on a pin by pin basis which is useful in the case of the UNKNOWN_LOADING property attached to the body of a library part.

8.10 ASSIGNING LIBRARY PIN NUMBERS

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.
- What the pin numbers are for each section.

The Packager will print an error message if a pin is found with no PIN_NUMBER property.

The basic form for a PIN_NUMBER property is:

PIN_NUMBER = (<pin number>)

where <pin number> is a positive integer. If the pin represents a vector (multiple bits) rather than a scalar (single bit), the pin numbers for the pin are specified as:

PIN_NUMBER = (< <pin number>, <pin number>, ... >)

The enclosing '<' and '>' serve to indicate that the pin represents multiple bits. The pin numbers in the list must separated by commas. For example, a four bit pin might be specified as:

PIN_NUMBER = (<1,2,4,5>)

If a part has multiple sections, the PIN_NUMBER must specify the pin numbers for each section. The form of the PIN_NUMBER property for specifying sections is:

PIN_NUMBER = (<pin number>, <pin number>, ...)

where <pin number> specifies the pin number for the same pin but for different sections. For example, the output pin of a 74LS00 (a quad NAND) would be specified as:

PIN_NUMBER = (3,6,8,11)

There must be four pin numbers specified since the part has four sections. All pins of the part must be assigned the same number of pin numbers (indicating the number of sections). The Packager will print an error message if this is not so.

If a pin is common to each of 4 sections, it must be given 4 pin numbers as well; the pin numbers are all identical. For example, the clock and Q pins of a 74LS273 (an octal register) would be specified as follows:

```
PIN_NUMBER = (2,5,6,9,12,15,16,19)      Q pin
PIN_NUMBER = (11,11,11,11,11,11,11,11)  clock
```

Note that the clock pin has 8 identical entries because it is common (has the same pin number) for each section of the part.

Care must be taken to ensure that the pin numbers are consistent for all pins of each section. Each number in the list specifies a different section. The Packager expects the second number in the list, for example, to correspond to the second section for every pin of the part.

If a sectioned part has a vectored pin, its pin numbers are specified in a similar manner. For instance, a 3 bit pin in a part with 2 sections might be specified as:

```
PIN_NUMBER = (<1,2,3>, <5,6,7>)
```

8.11 THE LOCATION PROPERTY

The LOCATION property is attached to a body in a drawing to assign its physical part designator. LOCATION properties can be attached only to physical part bodies. LOCATION properties attached to higher level drawings are errors and ignored. The LOCATION property is not inherited as a body property.

The LOCATION property always takes precedence over a physical part designator assignment in the physical part designator transformations file. An attempt at reassignment is flagged as an error. This error is classified as a FATAL ERROR. The Packager outputs a list of such discrepancies so that the drawings can be altered.

The LOCATION property is to be used only for flat drawings or wherever there is a one to one correspondence between a body in a drawing and a physical part. Since the LOCATION property specifies the physical part designator, care should be taken to make sure that layout has been considered.

Several parts may be given the same LOCATION property as long as they may all be assigned (as sections) to the same physical part. If this is not the case, an error message is produced.

8.12 THE LOCATION_CLASS PROPERTY

The LOCATION_CLASS property is used to control the assignment of logical parts to physical part by the Packager. If two logical parts have different LOCATION_CLASS properties, they will not be assigned to the same physical part. However a logical part without a LOCATION_CLASS may be assigned to a physical part that already has a logical part with a LOCATION_CLASS. LOCATION_CLASS properties are attached as body properties in the drawings.

8.13 MANUAL SECTION ASSIGNMENTS

The user can manually assign sections to logical parts in the drawings and have the Packager perform the specified assignments. Sections are assigned through the Graphics Editor SECTION command and works much like the VERSION command by pointing to the body or pin of the logical part.

Currently, the only parts that can be assigned to a particular section are either SIZE wide parts with a size of 1 or HAS_FIXED_SIZE parts. Assigning sections to a HAS_FIXED_SIZE part is accomplished by pointing to the pin of the section to be assigned. It is an error to point to the body of a HAS_FIXED_SIZE part.

If the logical part selected can be assigned to a section, the pin numbers for the selected section will be back annotated to the part in the drawing. If the same logical part is selected again, the next section will be selected and the new pin numbers will be back annotated to the part. Thus by pointing to the same logical part, one can step through all the different possible sections for the logical part.

The actual implementation for section assignment is done through the use of the SEC property which is assigned by the Graphics Editor to the logical part. If the Packager finds this property on a logical part, it will assign the logical part to the desired physical section. The user should not use or change the SEC property assigned by the Graphics Editor.

8.14 MANUAL PIN ASSIGNMENTS

The user can manually assign pins of a logical part in the drawings and have the Packager perform the specified assignments. The pins are assigned through the Graphics Editor command PINSWAP.

Currently, the only parts that can have pin assignments are those which already have been assigned to a section through the Graphics Editor command SECTION. It is an error to try and PINSWAP pins of a part which has not been SECTIONed.

In addition, only pins in the same swap group are permitted to be swapped. 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 an NAND gate like 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.

To define a swappable group, the library files must have the PIN_GROUP property defined. Any set of pins that is swappable must have the PIN_GROUP attached to it with the same value. Any pin without the PIN_GROUP property is considered not swappable with any other pins. The value of the PIN_GROUP property is not important, only that all pins of a swappable group have the identical value.

Once pin swaps have been performed on a part, further section assignments are no longer allowed for the part. This means that if the user wishes to assign a part to a different section after performing pin swaps, the part must first be de-assigned by using the REPLACE command. The user can then assign the new part to the desired section.

The actual implementation for pin assignment is done through the use of the PN property which is assigned by the Graphics Editor to the pins of the logical part. If the Packager finds this property on a pin, it will assign the nodes to the desired physical pin. The user should not use or change the PN property assigned by the Graphics Editor.

8.15 THE CREATION OF PHYSICAL NAMES

The Packager assigns physical names to both signals and parts the first time through. The algorithms are described below.

CREATION OF SIGNAL NAMES

Physical net names are created from the abbreviation of the logical signal name for the net. The maximum length of the names is controlled by the NET_NAME_LENGTH directive

always. 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 characters of the name. This is to preserve the bit offset that was appended to the name.

CREATION OF PHYSICAL PART DESIGNATORS

Physical part designators are created by starting with the value of the PHYS_DES_PREFIX property found on the part type in the library, or if no such property is found, the standard prefix 'U'. If the name is not unique, it is made unique by suffixing a number. The maximum length of the names is controlled by the PART_NAME_LENGTH directive.

8.16 FEEDBACK PROCESSING

The Packager converts a logical design into a format suitable for physical design. This design as output by the Packager may not be optimal for layout, and the physical design system may rearrange parts and swap equivalent sections within a part and equivalent pins on a section. Since users need documentation of the completed physical design and may make modifications which require changes to the physical design, any changes to the physical design made during layout and wiring must be fed back into the Packager state files.

There are 4 types of changes which are commonly made during physical design:

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. Example G13 is row G column 13.

2. Physical section reallocation.
   To simplify wiring, it is often desirable to group together those parts which 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 into 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 in the form of swaps of two sections between different parts of the same type. Sections are sometimes reallocated 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 reallocated. This is done primarily to parts having many equivalent inputs such as a 13 input NAND gate.

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 documentaion or standardization reasons.

The Packager currently can process four types of feedback files. The user has 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. The state files should never be edited by the user. These files are as follows:

1. PSTPRTX - Physical part designator transformations file
   This file contains a list of old physical part designator and new physical part designator pairs.

2. PSTSECX - Physical section transformations file
   This file contains a list of old physical section to new physical section pairs.

3. PSTNETX - Physical net name transformations file
   This file contains a list of old physical net name to new physical net name pairs.

4. PSTFNET - Feedback net list
   This file contains a net list sorted by physical part designator of the current physical design. The only restriction is that the physical net name must not have changed or the new physical net names have already been fedback before performing a net list feedback. The Packager will detect and perform all physical part designator changes, physical section swapping, and pin swapping from the feedback net list.

The FEEDBACK_ORDER directive is used to specify which files and order of feedback processing for the Packager. The file types allowed are as follows:

PART_TRANS
: This specifies the physical part designator transformations file (PSTPRTX).

SECTION_TRANS
: This specifies the physical section reallocation file (PSTSECX).

NET_TRANS
: This specifies the physical net name transformations file (PSTNETX).

FEEDBACK_NETLIST
: This specifies the feedback net list (PSTFNET).

An example of this directive is as follows:

```
FEEDBACK_ORDER  NET_TRANS, FEEDBACK_NETLIST;
```

which specifies that physical net name transformations occur first followed by a feedback net list transformation.

Feedback of physical design changes can only occur when state files generation have been enabled, otherwise the changes being fedback cannot be saved. The Packager will generate an error if the FEEDBACK_ORDER directive is used without enabling the use of state files through the USE_STATE_FILES ON; directive.

Due to the current implementation, feedback processing will only work if the logical design (the compiler expansion file) has not changed since the design was last packaged to generate a physical design from which the changes are derived and the design now being processed for feedback changes. This means the compiler expansion file, library files, and all Packager generated state files should be saved for a design that is sent to a physical design system. By saving these files, the Packager will be able to feedback the physical changes made by the physical design system.

8.17 BACK ANNOTATION

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

The most common form of back annotation is bringing information from the Packager (physical design information) and adding it to the drawings. The Packager generates a back annotation file that contains physical information grouped by drawing. To generate this file, the directive OUTPUT BACKANNOTATION; should appear in the directive file. Backannotation can occur on three types of elements: bodies, pins, and nets. In order to select among these elements, the ANNOTATE directive should be used. If this directive is not specified, the default options will be bodies and pins. If the net option is specified, the synonym file from the compilation must be available. The back annotation information is written to the logical file PSTBACK.

The back annotation file can then be read by the Graphic Editor with the BACKANNOTATE command. This will add all the physical information in the file to the drawings. It is recommended that you copy all the drawings to another directory before back annotation is performed. This will give you both non-annotated and annotated versions of the drawings which may be useful since it is not easy to remove the annotated properties from a set of drawings.

8.18 OUTPUT FILE FORMATS

This section describes the formats of each of the output files created by the Packager. All output files are text files.

STATE FILES

The Packager generates several state files to maintain the physical assignments through several runs of the Packager. The state files are written to the logical files PSTPRTB, PSTSIGB, PSTPSWP, and PSTSTAT. They are described in the section "Packager State Files" later in this chapter.

CROSS REFERENCES

The Packager generates several cross references to the logical file PSTXREF. A complete description of these files and their use is described in the "Packager Cross References" section later in this chapter.

EXPANDED NET AND PART LISTS

The expanded net and part lists produced by the Packager contain logical to physical net and part bindings, as well as all the properties attached to pins and bodies. These files are intended for use in interfacing to unsupported physical design systems. These list are written to the logical files "PSTXNET" and "PSTXPRT" respectively. They are described in the "Packager Expanded Lists" section later in this chapter.

LOGICAL CHANGES LIST

This file consists of a list of the changes in the logical design between the current input and the last run of the Packager. The changes are essentially add and delete lists of logical parts.

The Packager compares the Compiler expansion file and the Packager state files to find the following:

1. Any logical parts that were not present during the last run of the Packager.

2. Any logical parts present during the last run of the Packager and are not present during the current run. Each of these parts has a corresponding physical part designator.

The Packager lists each of the parts found to give a summary of the changes made. The form of the list is:

```
<header>
LOGICAL PARTS ADDED TO DESIGN:
  <list of logical part designators>
LOGICAL PARTS DELETED FROM DESIGN:
  <list of logical part and physical part designators>
END LOGICAL CHANGES LIST
```

where <list of logical part designators> lists each logical part designator separated by ';'. The second list consists of a logical part designator followed by a physical part designator separated by ' ' and terminated with a ';'.

An example Logical Changes file:

```
LOGICAL CHANGES LIST - 1  12-AUG-1982 13:18:10.21
LOGICAL PARTS ADDED TO DESIGN:
  (FG1 .HH 1.TTT .00)74LS00;
  (ABC XYZ .253)74LS253;
LOGICAL PARTS DELETED FROM DESIGN:
  (ABC XYZ .122)74LS122 U31;
  (REG .04)74LS04 U76;
END LOGICAL CHANGES LIST
```

8.19 POWER AND GROUND PIN ASSIGNMENTS

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 form of the PIN_NUMBER property value is:

```
( <power rail> : <pin list> ; ... )
```

where <power rail> is the name of the power supply rail and must be an identifier (a string of letters, digits, or '' starting with a letter). The <pin list> is a list of the pin numbers of the part (separated with commas) that connect to the power rail. The ';' is used to separate the power rail specifications.

For example, a TTL part has two power pins: VCC and GND. The 74LS00 would have the property:

```
POWER_PINS = (VCC:14; GND:7)
```

The TMS4050 RAM would have the following property:

```
POWER_PINS = (VBB:1;VDD:10;VSS:18)
```

The 100123 bus driver would have the following property:

```
POWER_PINS = (VCC:6; VCCA:7,9,11,5,3,1; VEE:18);
```

The order of the values is not important. The POWER_PINS property only applies to parts found within the libraries and is ignored if found elsewhere.

8.20 DESCRIPTION OF THE CHIPS FILE

The CHIPS file contains a description of every physical part in the libraries. It is generated by the Compiler from a drawing that uses every physical part exactly once. The Valid supplied libraries have such drawings. See, for example, the LSTTL LIBRARY drawing in the LSTTL library.

The CHIPS file contains information attached to the body as well as the .PART drawing describing the part. The following information is expected for each part:

1. PIN_NUMBER property on every pin.
2. INPUT_LOAD or OUTPUT_LOAD properties on every pin.
3. POWER_PINS property for the part. This may be attached to the body or to the .PART drawing (attached to the DRAWING body within the .PART drawing).
4. FAMILY property for the part. This may be attached to the body or to the .PART drawing (attached to the DRAWING body within the .PART drawing).

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.

The CHIPS file is generated with the Compiler by compiling the library description drawings with the OUTPUT CHIPS; directive specified. The Compiler produces a chips file in the file CHIPS which is read by the Packager. 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 libraries are described, for the purpose of creating the CHIPS file, in drawings of the form: <library> LIBRARY where <library> is the name of the particular library. For example, the LSTTL library is described in the drawing LSTTL LIBRARY. The library description drawing contains exactly one instance of each of the parts in the library. Only one version of parts with multiple body versions is permitted in the library description drawing. If the part has body versions that are asymmetrical (for example if the two versions describe sections of the part that have different function) both versions of the part must appear.

8.21 FORMATS FOR USER GENERATED FILES

This section describes the formats of the files you can generate. The Packager reads these files. All input files for the Packager are text files. Every file is terminated by an 'END.' which serves to mark the end of the file as well as provide a method for determining whether the file is complete.

The header lines in the file serves to identify the file and the name of the design. The form of the header lines are:

```
FILE_TYPE = <file type> ;
ROOT_DRAWING = '<drawing name>' ;
```

where <file type> specifies the file's type (see below for each file described) and <drawing name> is the name of the root drawing of the expansion file. The header for the Physical Part Designator Transformations file would appear as:

```
FILE_TYPE = PART_TRANS;
ROOT_DRAWING = 'RISC/E II';
```

Comments may be placed in the files if enclosed in '{' and '}'. A comment may appear anywhere a space may appear. Comments may cross line boundaries. Comments 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 ('~') should be 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 is only significant if it occurs at the end of the line.

PHYSICAL PART DESIGNATOR TRANSFORMATIONS

This file is used to change a physical part designator assigned by the Packager to one determined by the user. The physical part designator is used to identify a particular instance of a physical part. Each physical part is (usually) a real, purchaseable, wireable, tangible entity (unlike Valming Verifier or Simulator primitives).

The file type for this file is PART_TRANS. The file consists of a list of transformations. Each transformation is of the form:

```
'<old part designator>' '<new part designator>'
```

where <old part designator> is the physical part designator assigned by the Packager during its last run and <new 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 = 'RISC/E II';
```

```
'U31' 'U32'
END.
```

PHYSICAL NET NAME TRANSFORMATIONS

This file is used to change the name given a physical net. Each net is originally assigned a name by the Packager. That name can be changed with this file. The file type for this file is NET_TRANS. The file consists of a list of transformations. Each transformation is of 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 with the transformation:

```
FILE_TYPE = NET_TRANS;
ROOT_DRAWING = 'RISC/E II';
'N00001' 'XYZ'
END.
```

PHYSICAL SECTION TRANSFORMATIONS

The Packager assigns sections during the first run of a design. Sections of parts are assigned sequentially so that individual bits of a signal will be connected to the same package. No layout knowledge is used during this 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.

The file contains a list of old physical pin designators (as assigned by the Packager during its last run) and new physical pin designators where a physical pin designator consists of a physical part name and a unique pin number of the section (not a common pin). The Packager will reassign the sections as specified in this file. If a section is reassigned that was already assigned in the drawings (through section assignment), the Packager will NOT change the assignment. An error message is printed in this case. This error is classified as a FATAL ERROR. The only way to change section assignments assigned in the drawings is to change the drawings.

The file type for this file is SECTION_TRANS. The file consists of a list of transformations. Each transformation is of the form:

```
'<old part>' <old pin> '<new part>' <new pin>
```

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

```
FILE_TYPE = SECTION_TRANS;
ROOT_DRAWING = 'RISC/E II';
'U31' 1 'U31' 4
'U31' 4 'U31' 1
END.
```

FEEDBACK NET LIST

If a physical design system can generate a net list which specifies part types and represents a physical design differing from the design produced by the last run of the Packager only by physical part designator changes and physical section reallocation, then the Packager can extract these changes and perform these transformations. The Feedback Net List is used by the Packager to extract these transformations of physical designators and section assignments.

The file type for this file is FEEDBACK_NETLIST. The file consist of a list of nodes of the design. Each node entry is of the form:

```
'<physical part name>'
'<physical part type>'
<pin number>
'<physical net name>' ;
```

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

The file MUST be SORTED by physical part name, 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 = 'RISC/E II';
'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.
```

8.22 INTERFACE SIGNALS

Interfaces between a circuit and the external components it connects to must be defined in some manner. This is normally done in board level designs by implicitly defining interface signals through connectors spread throughout the design. Many gate array design systems, however, treat interface signals (those which connect to the chip carrier and hence go off the chip) as different 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 specially.

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 to the interface signals the IO_NET property with either the values INPUT, OUTPUT, or BIDIRECTIONAL, as defined by the FLAG body. These properties are then output in the Expanded Net List.

Packager Directives Summary

8.23 PACKAGER DIRECTIVES

The Packager directives are used to specify input and output files, control message generation, and direct the Packager execution. The directives are placed in a text file and given to the Packager as the logical file INFILE which is bound to the file PACKAGER.CMD (PACKAGER CMD in CMS) by default. Each of the directives is described below. The directives and their parameters are not case sensitive.

ANNOTATE

Used to specify the allowable schematic back annotation information to generate for the BACKANNOTATION output file. If more than one of these directives appears in the directive file, the Packager ignores all but the last one. The form of the directive is:

ANNOTATE <option> , <option> , ... ;

The options for this directive are as follows:

BODY
When specified, allows the back annotation of physical part designators.

PIN
When specified, allows the back annotation of physical pin numbers.

NET
When specified, allows the back annotation of physical net names. Currently only scalar nets can be annotated. The synonym file from the compilation of the design must be available for the net option to be successful.

If directive is unspecified, the Packager will generate back annotation for BODY and PIN when the BACKANNOTATION output file is generated.

FEEDBACK_ORDER

Used to specify which feedback files and their order to perform feedback processing for the Packager. This

directive can only be used in the directives file once. The form of the directive is:

```
FEEDBACK_ORDER <file type>, <file type>, ... ;
```

The allowable <file type> for this directive are as follows:

PART_TRANS
: This specifies the physical part designator transformations file (PSTPRTX).

SECTION_TRANS
: This specifies the physical section reallocation file (PSTSECX).

NET_TRANS
: This specifies the physical net name transformations file (PSTNETX).

FEEDBACK_NETLIST
: This specifies the feedback net list (PSTFNET).

An example of this directive is as follows:

```
FEEDBACK_ORDER  NET_TRANS, FEEDBACK_NETLIST;
```

which specifies that physical net name transformations occur first followed by a feedback net list transformation. If this directive is unspecified, the Packager will not perform feedback processing.

FILTER_PROPERTY

Used to specify properties that are not to be included in the expanded net and part lists. A list of property names separated by commas is given. The Packager makes sure that these properties are not output to the expanded net and part lists. This directive has no impact on any other file. The properties FOO, GRBX, and PATH could be suppressed with the directive:

```
FILTER_PROPERTY FOO,GRBX,PATH;
```

Any number of properties can be listed. The FILTER_PROPERTY directive can be specified as many times as desired.

INCLUDE_IO_LIST

Used to control whether the nets connected to the interface pins of the design are included in the Expanded Net List with the IO_NET property. The value of the IO_NET property is either INPUT, OUTPUT, or BIDIRECTIONAL as defined by the FLAG body. The directive is specified as follows:

INCLUDE_IO_LIST ON;	output the IO_NET property for the interface pins.
INCLUDE_IO_LIST OFF;	do not output the IO_NET property for the interface pins.

If this directive is unspecified, the Packager will not output the interface pins.

LIBRARY_FILE

Used to specify the names of files containing library components. These files are produced by the Compiler using the OUTPUT CHIPS directive. Any number of libraries can be specifed with this directive. The names can be placed in a list separated by commas or listed individually with separate LIBRARY_FILE directives. For example, the directive:

```
LIBRARY_FILE '100k.prt', 'lsttl.prt';
```

specifies two library files, 100k.prt and lsttl.prt, and is equivalent to the directives:

```
LIBRARY_FILE '100k.prt';
LIBRARY_FILE 'lsttl.prt';
```

The Packager checks to make sure that a file is not specified more than once.

MAX_ERRORS

Used to specify the maximum number of errors allowed before the Packager gives up and terminates. When this condition occurs, the Packager prints a message and terminates with a summary of the execution. The maximum number of errors can be set to 500 as follows:

```
MAX_ERRORS 500;
```

If not specified, the Packager terminates after 1000 errors.

OUTPUT

Used to control which output files are produced by the Packager. Each of the various output listings can be individually suppressed or enabled. All files are generated by default. The first OUTPUT directive encountered causes all output files to be turned off (so that they may be individually turned back on) unless the '-' option is used, in which case files are deleted individually. The 'ALL' identifier can be used to turn all files on or off. For example, the directive:

```
OUTPUT;
```

is equivalent to the directive

```
OUTPUT -ALL;
```

which turns off all output files.

The names of these files are listed separated by commas in a single OUTPUT directive or can be specified with multiple OUTPUT directives. For instance, the directive:

```
OUTPUT EXPANDEDNETLIST,EXPANDEDPARTLIST;
```

is equivalent to:

```
OUTPUT EXPANDEDNETLIST;
OUTPUT EXPANDEDPARTLIST;
```

In both of the above examples, the only output files that will be generated are the expanded net list and the expanded part list.

Each of the OUTPUT files are listed below.

EXPANDEDNETLIST
: Causes the expanded net list to be output to the file PSTXNET.

EXPANDEDPARTLIST
: Causes the expanded part list to be output to the file PSTXPART.

LOGICALCHANGES
: Causes the logical changes summary to be output to the

file PSTLCHG.

CROSSREFERENCES
Causes all of the cross references to be output to the file PSTXREF.

LOCALPARTXREF
Causes the local part crossreference to be output to the file PSTXREF.

GLOBALSIGNALXREF
Causes the global signal crossreference to be output to the file PSTXREF.

GLOBALPARTXREF
Causes the global part crossreference to be output to the file PSTXREF.

BACKANNOTATION
Causes the back annotation file to be output to the file PSTBACK.

If no OUTPUT directive is specified, the Packager produces all files.

OVERSIGHTS

Used to control whether oversight messages are displayed. Several conditions are detected during execution of the Packager that are considered to be more serious than a warning (see below) but not as serious as an error. The oversights should be corrected, but the design will probably work without first fixing them. The total number of oversights detected is always reported at the end of the program regardless of whether they were printed or not. This directive is used to turn off all oversight messages. The SUPPRESS directive can be used to turn off specific oversight message by message number. The directive is specified as follows:

OVERSIGHTS ON;	display all oversight messages on the Packager's list file.
OVERSIGHTS OFF;	display no oversight messages.

PART_NAME_LENGTH

Used to control the maximum physical part name designator length to be generated by the Packager. The form of the directive is:

```
PART_NAME_LENGTH <length> ;
```

If this directive is unspecified, the Packager will use a default length of 16.

PART_TABLE_FILE

Used to specify the names of files containing physical part tables. Any number of physical part table files can be specifed 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';
```

The Packager checks to make sure that a file is not specified more than once.

PASS_PROPERTY

Used to control whether a specific property appears in the expanded net and part lists. The PASS_PROPERTY directive takes a list of properties like the FILTER_PROPERTY directive above. If the directive is not specified, all the properties are defaulted to pass into the expanded lists. Once specified, only those explicitly selected properties are allowed to pass into the expanded lists. The pass operation is performed before filtering, thus those properties explicitly allowed to pass may then be suppressed by filtering them out. See also the FILTER_PROPERTY directive.

NET_NAME_LENGTH

Used to control the maximum physical net name length to be generated by the Packager. The form of the directive is:

NET_NAME_LENGTH <length> ;

If this directive is unspecified, the Packager will use a default length of 24.

REPORT

Used to control which user reports to generate. The syntax of this directive is the same as the OUTPUT directive and supports the '-' and 'ALL' options. All the reports are written to the logical file PSTRPRT. The available reports are as follows:

SPARES
This report contains the all the spare physical sections in the design. Spare physical sections are sections which have not been allocated to a logical part and are listed by physical part designator and a unique pin number of the section that is a spare.

PARTSUMMARY
This report contains a summary of all the physical parts used in the design. It is a list of part types and the number of physical parts used for that part type. A grand total is also generated for all the physical parts used in the design.

If no REPORT directive is specified, the Packager produces all reports.

SUPPRESS

Used to suppress specific warning and oversight messages. Warnings and oversights are used to grade the severity of error conditions detected by the Packager. Warnings are considered to be the least severe, followed by oversights, and then errors. Since neither warnings nor oversights are as severe as an "error", and since there may be many of these messages in a good design, this directive is supplied to suppress the message that would be produced. The design conventions assumed by the

Packager are conservative and rigorous. The user may choose to design in a more liberal style and may want to ignore certain messages. Warning 132 can be suppressed with the directive:

```
SUPPRESS 132;
```

A list of warning messages may be specified as, for instance:

```
suppress 132,133,134;
```

All warning messages can be suppressed with the WARNING directive (see below). Error messages cannot be suppressed. If unspecified, the Packager suppresses no warnings or oversights.

USE_STATE_FILES

Used to control whether the Packager reads and generates state files. The directive is specified as follows:

`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.

If this directive is unspecified, the Packager will use and generate state files.

WARNINGS

Used to control whether the Packager prints warning messages. Several conditions are detected that are not as severe as errors, but need to be brought to the attention of the designer. All warning conditions can be eliminated by adding the needed information (described in the warning message) to the drawings. This directive can be used to suppress all warning messages (though it is a good idea to add the information to the drawings - this info helps to more clearly document the design). The total number of warning conditions encountered is reported at the end of the program regardless of whether warnings are displayed or not. The directive is specified as follows:

`WARNINGS ON;`	display all warning messages on

the Packager's list file.

WARNINGS OFF; display no warning messages.

If unspecified, the Packager prints all warning messages.

8.24 AN EXAMPLE OF A PACKAGER DIRECTIVES FILE

The Packager directives file can be created with a text editor. The Packager pays no attention to the end-of-line or to multiple spaces. The letter case of the directives is unimportant. This is true both for directive names as well as file names within strings. Comments may be placed in the file if enclosed with '{' and '}'. Note that all Packager directives must be separated by ';' and the file must end with an 'end.'.

```
output EXPANDEDNETLIST,  { output the expanded net list     }
       EXPANDEDPARTLIST; { output the expanded part list    }
warnings on;             { display all warning messages     }
end.                     { this marks the end of the file   }
```

HARD-LOC-SET OFF;

The PIN_NUMBER Property

8.25 INTRODUCTION

The Packager recognizes the PIN_NUMBER property on pins of parts in the CHIPS files. The property is used to assign the physical pin numbers for each logical pin so that the Packager can do physical assignments. The Packager requires PIN_NUMBER properties for every pin of every part. The absence of the property on any pin is considered fatal.

The PIN_NUMBER property for scalar pins has the following form:

```
PIN_NUMBER = ( <pin>, <pin>, ... )
```

where <pin> is a pin number. A multiple section part has several pin numbers for a pin, one for each section.

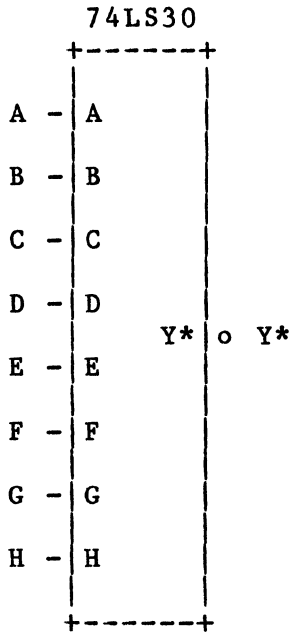
A PIN_NUMBER property on a vector pin has the following form:

```
PIN_NUMBER = ( < <pin>, <pin>, ... >,
               < <pin>, ... >, ... )
```

where <pin> is a pin number which can be either an integer or an identifier consisting of letters, digits, or '_' with a maximum length of 16 characters. The enclosing '<' and '>' are used to indicate that the list of pin numbers specifies individual bits of the pin and not different sections for the pin.

8.26 PIN_NUMBERS FOR SINGLE SECTION SCALAR PINS

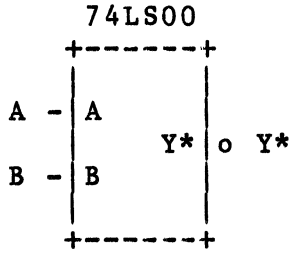
A scalar pin is a pin which corresponds to a one bit signal. This is different from a vector pin which corresponds to a several bit signal. The PIN_NUMBER property for each pin of a simple one section part like a 74LS30 is a single positive integer enclosed by parentheses as follows:



PIN	PIN_NUMBER property
A	(1)
B	(2)
C	(3)
D	(4)
E	(5)
F	(6)
G	(11)
H	(12)
Y*	(8)

8.27 PIN_NUMBERS FOR MULTIPLE SECTION SCALAR PINS

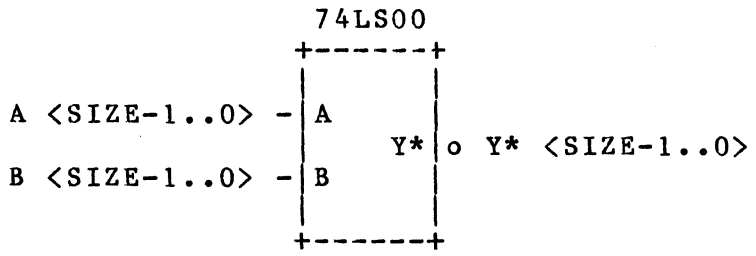
Some physical parts contain several equivalent sections which each correspond to the logical part of the given type. An example is the 74LS00, which contains 4 2-input NAND gates, and the logical part for a 74LS00 is a single 2-input NAND gate. The sections of a part are specifed by the PIN_NUMBER property on a logical pin by logical pin basis. Each pin of the logical part has a PIN_NUMBER property containing a list of pin numbers, one pin number for each section in the part. The sections are ordered, with all pins for the Nth section present in the Nth position in the PIN_NUMBER property values. Each logical pin of a 74LS00 has 4 pin numbers, one for each section. The pin numbers for various sections within the PIN_NUMBER property are separated by commas as follows:



PIN	PIN_NUMBER property
A	(1,4,9,12)
B	(2,5,10,13)
Y*	(3,6,8,11)

8.28 PIN_NUMBERS FOR SIZE EXPANDED PARTS

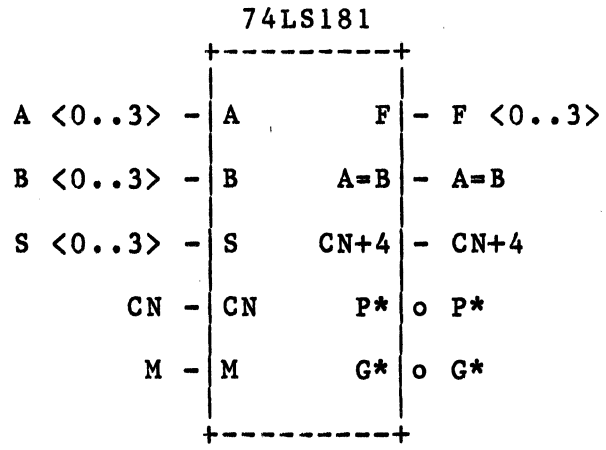
A scalar pin may be defined as a SIZE wide pin on a SIZE replicated part, for which each bit of the attached bus connects to a different section of the part. The part definition for a SIZE wide part defines the SIZE wide pins as scalars, since for each section, those pins are only one bit wide. The 74LS00 shown above could (and probably would) be defined as a SIZE replicatable part as follows:



PIN	PIN_NUMBER property
A <SIZE-1..0>	(1,4,9,12)
B <SIZE-1..0>	(2,5,10,13)
Y* <SIZE-1..0>	(3,6,8,11)

8.29 PIN_NUMBERS FOR VECTOR PINS

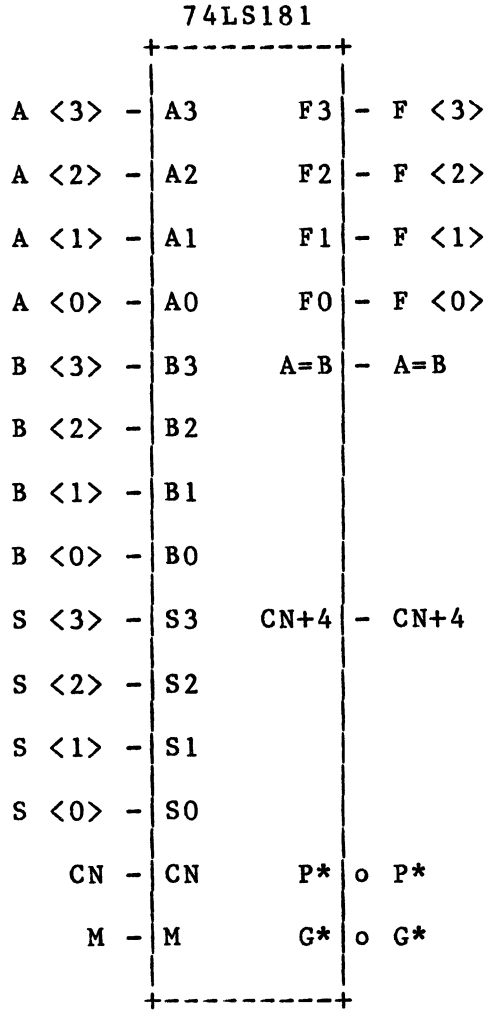
A vector pin is a multiple bit pin having a fixed number of bits (not effected by the value of the SIZE property), where each bit of the vector connects to the same section of the part. Examples of vector pins are the data buses of a 74LS181 ALU. A vector pin has a PIN_NUMBER property of the same form as a scalar pin, except that each logical pin number is generalized to include several physical pin numbers, enclosed between a less-than sign "<" and a greater-than sign ">":



PIN	PIN_NUMBER property
A <0..3>	(<19,23,21,2>)
B <0..3>	(<18,20,22,1>)
S <0..3>	(<3,4,5,6>)
CN	(7)
M	(8)
F <0..3>	(<13,11,10,9>)
A=B	(14)
CN+4	(16)
P*	(15)
G*	(17)

The bits of a vector pin are assigned to the physical pin numbers specified in the PIN_NUMBER property in the following manner: The bit having the lowest subscript is assigned to the first pin number in the list. The bit with the next lowest subscript is assigned to the second pin in the list, and so on. This is intuitively correct for installations using left-to-right bit ordering (example A<0..31>), since the pin numbers for vector pins are then ordered from most significant to least significant from left to right. An example of left-to-right bit ordering is shown for the 74LS181 above. For installations using right-to-left bit ordering (example B<15..0>), the assignment seems backwards, since the pin numbers for the vector pins are then ordered from least significant to most significant from left to right.

The way to define the pin numbers for vectors without using the vector form of the PIN_NUMBER property is to define an expanded body for each vectored part. Attach the PIN_NUMBER property to each bit of a vector pin separately as shown for the right-to-left bit ordering version of the 74LS181.

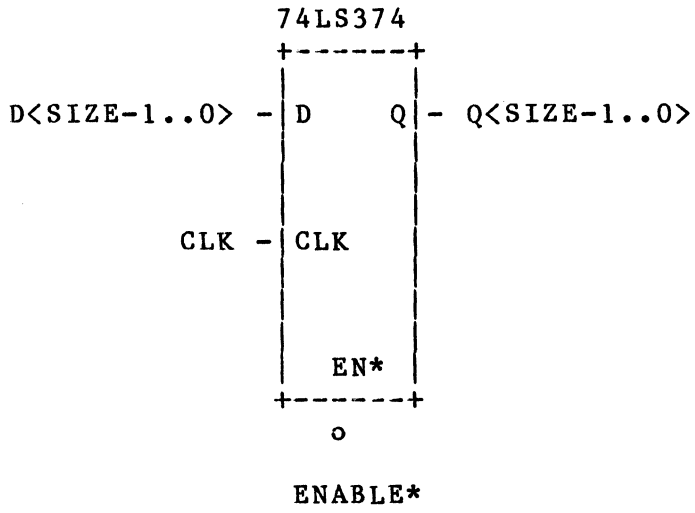


PIN	PIN_NUMBER property
A <3>	(19)
A <2>	(21)
A <1>	(23)
A <0>	(2)
B <3>	(18)
B <2>	(20)
B <1>	(22)
B <0>	(1)
S <3>	(3)
S <2>	(4)
S <1>	(5)
S <0>	(6)
CN	(7)
M	(8)
F <3>	(13)
F <2>	(11)
F <1>	(10)
F <0>	(9)
A=B	(14)
CN+4	(16)
P*	(15)
G*	(17)

A vectored version of the part may still be defined and used, as long as the non-vectored version is the one from which the CHIPS file is generated, and the pin names for the two versions are the same.

8.30 PIN_NUMBERS FOR PARTS WITH COMMON PINS

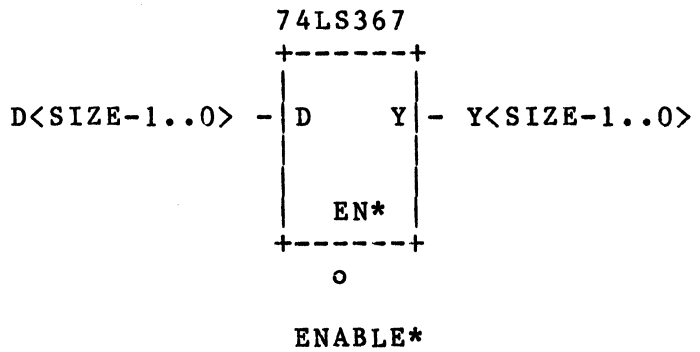
Some multiple section parts have pins which are common to several sections. A example of this is the 74LS374 octal register, which has 8 sections with a common clock and a common output enable. The PIN_NUMBER properties for these pins are the same as for any other pin, except that pin numbers appear more than once. In fact, common pins are treated EXACTLY like non-common pins, in that all pins for the Nth section of the part are present in the Nth position in the PIN_NUMBER property values. The 74LS374 would be defined as follows:



PIN	PIN_NUMBER property
D<SIZE-1..0>	(3,4,7,8,13,14,17,18)
Q<SIZE-1..0>	(2,5,6,9,12,15,16,19)
CLK	(11,11,11,11,11,11,11,11)
ENABLE*	(1,1,1,1,1,1,1,1)

The CLK and ENABLE pins are common to all eight flip-flops in the package. The D and Q pins are defined so that one bit is assigned to each flip-flop.

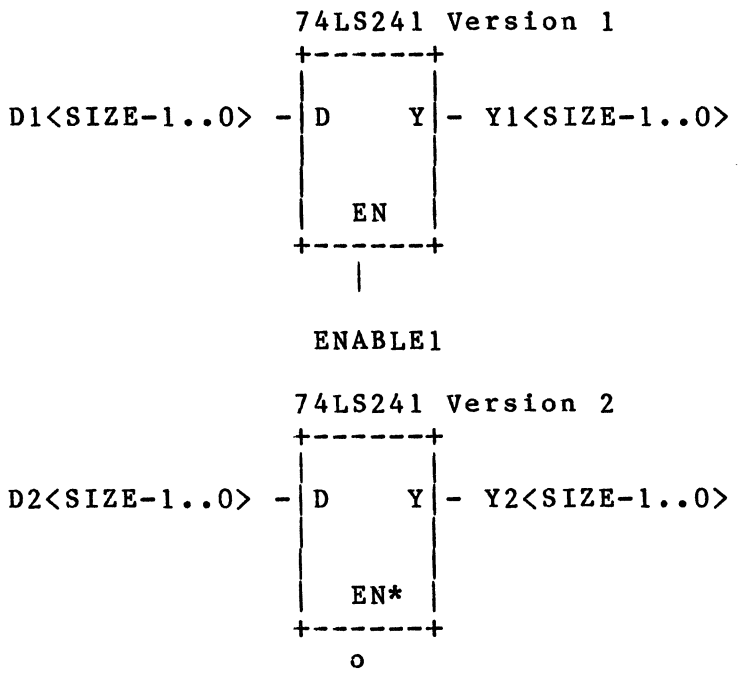
Some parts have pins which are common only to certain sections of the part. These are represented in the same manner as pins which are common to all sections, except that the these pin numbers are present only in the sections for which they are common. An example is the 74LS367:



PIN	PIN_NUMBER property
D<SIZE-1..0>	(2,4,6,10,12,14)
Y<SIZE-1..0>	(3,5,7,9,11,13)
ENABLE*	(1,1,1,1,15,15)

8.31 PIN_NUMBERS FOR PARTS WITH ASYMMETRICAL SECTIONS

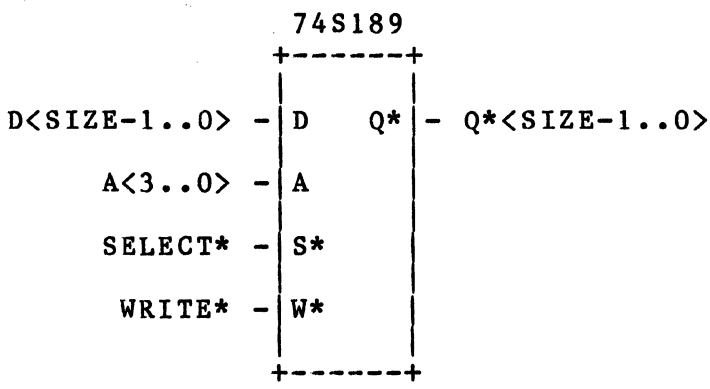
Some parts have multiple sections which are functionally different, such as the 74LS241, which has 4 buffers with active-high enables, and 4 buffers with active-low enables. One version of the body is defined for each type of section in the part. The pins of the different versions ALL have DIFFERENT pin names, so that a pin of a given name is present in only one section. The PIN_NUMBER property values for the pins specify all the sections of the part. Any pin which is not present in a given section is specified with a pin number of 0.



PIN	PIN_NUMBER property
D1<SIZE-1..0>	(11,13,15,17,0,0,0,0)
Y1<SIZE-1..0>	(9,7,5,3,0,0,0,0)
ENABLE1	(19,19,19,19,0,0,0,0)
D2<SIZE-1..0>	(0,0,0,0,2,4,6,8)
Y2<SIZE-1..0>	(0,0,0,0,18,16,14,12)
ENABLE2*	(0,0,0,0,1,1,1,1)

8.32 VECTOR PINS COMPARED TO SCALAR PINS

Vector pins are syntactically identical to scalar pins; they may be used wherever a scalar pin may be used, provided the pin to which they attach is a vector. An example part having vector pins and multiple sections is the 74S189, 16 word by 4 bit RAM:



PIN	PIN_NUMBER property
D<SIZE-1..0>	(4,6,10,12)
Q*<SIZE-1..0>	(5,7,9,11)
A<3..0>	(<1,13,14,15>,<1,13,14,15>,<1,13,14,15>, <1,13,14,15>)
SELECT*	(2,2,2,2)
WRITE*	(3,3,3,3)

The PIN_GROUP Property

8.33 INTRODUCTION

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.

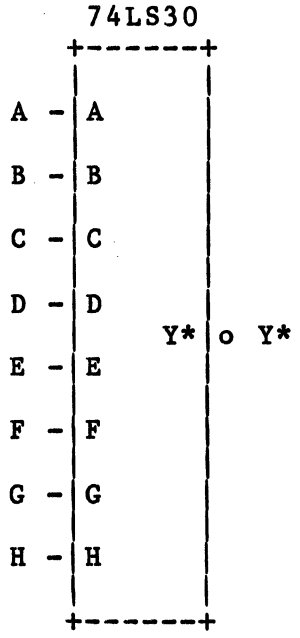
8.34 HOW TO USE THE PIN_GROUP PROPERTY

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 an NAND gate like 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 considered not swappable with any other pins. The value of the PIN_GROUP property is not important, only that all pins of a swappable group have the identical value.

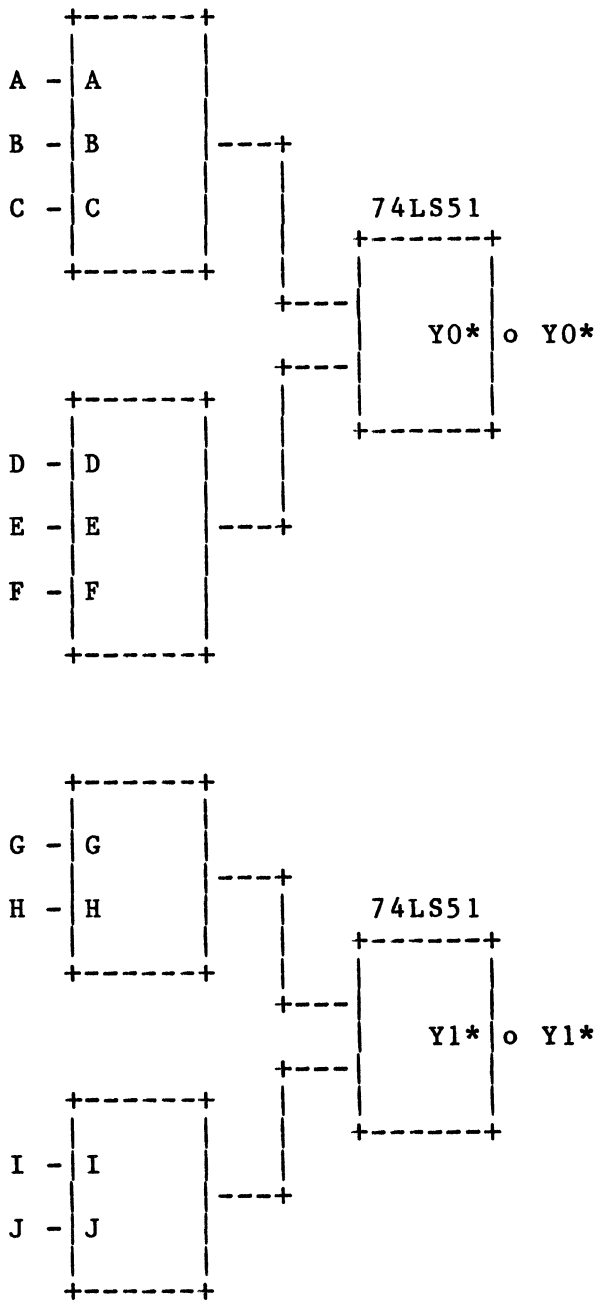
An example of pins which form a swappable group are the input pins of a 74LS30, an 8-input NAND gate. For this part, we can define the PIN_GROUP as follows:



PIN	PIN_GROUP property
A	1
B	1
C	1
D	1
E	1
F	1
G	1
H	1
Y*	not present on this pin

Notice in this case that the output pin Y* does not have a PIN_GROUP property because it is not swappable with any other pin of the part.

Another example of swappable groups appears for the 74LS51, a 2-wide 3-input, 2-wide 2-input AND-OR-INVERT. For this part, we can define the PIN_GROUP as follows:



PIN	PIN_GROUP property
A	1
B	1
C	1
D	2
E	2
F	2
Y0*	not present on this pin
G	3
H	3
I	4
J	4
Y1*	not present on this pin

In this case, there are 4 swappable groups, one for each AND gate. The inputs for each AND gate are equivalent and thus swappable. However, inputs from different AND gates are not swappable, so the PIN_GROUP properties have a different value.

Again note that the output pins Y0* and Y1* do not have PIN_GROUP properties since they cannot swap with any other pin.

Packager Expanded Lists Specifications

8.35 HOW ARE THE EXPANDED LISTS TO BE USED?

The Expanded Part List and Expanded Net List are intended to be used to interface to other systems. You can read these files, which contain all the information about the design known to the Packager, and convert them into a form compatible with some target system. The files are organized by physical information since that is the form needed when converting formats for other systems.

8.36 SOME GENERAL NOTES OF THE 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:

'<assertion><name>' <subscript>

where the <assertion> character is '-' and only appears for low asserted signals.

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. Comments may not be nested.

If an item is too long to fit on a line (80 characters), it will 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.

8.37 THE EXPANDED NET LIST

The Expanded Net List is a net list ordered by physical net name that contains all the information in the Concise Net List as well as all the properties known by the Packager for each net. This includes all the properties and the logical to physical binding of nets and nodes.

THE FORM OF THE EXPANDED NET LIST

The form of the Expanded Net List is as follows:

```
FILE_TYPE=EXPANDEDNETLIST;
<list of nets>
END.
```

The <list of nets> is of the form:

```
NET_NAME
<physical net name> <logical net name> <version> :
<net properties>
<node list>
```

where NET_NAME is used to mark the beginning of a net entry. <physical net name> is the physical name for the net. It is a quoted string of letters and digits. <logical net name> is the logical name for the net. It is in Valid canonical syntax. <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 <version> is not output. A colon (':') is used to mark the end of the logical net name. <net properties> are the properties of that net of 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, a semicolon is output alone.

The <node list> is a list of all of the nodes on the net and the properties for each. Each element has the form:

```
NODE_NAME
<physical designator> <pin number>
<logical node list>
```

where <physical designator> is the name of the physical part. It is a string of letters and digits. <pin number> is the pin number on that part.

The <logical node list> is a list of logical nodes corresponding to the physical node. Each element has the form:

<logical designator> <bit> <version> : <pin name> : <node properties>

where <logical designator> is the name of the logical part corresponding to the physical part. It is a quoted string of characters. <bit> is an optional field of the following form:

<bit number>

where <bit number> 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 number> is 0, then <bit> is not output. <version> is an optional field described above for <list of nets>. <pin name> is the logical name of the pin corresponding to the pin number given. It has Valid canonical syntax (see the previous 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.

AN EXAMPLE EXPANDED NET LIST

An example of an Expanded Net List:

```
FILE_TYPE=EXPANDEDNETLIST;
NET_NAME
'FOO2'                  { physical net FOO }
 -'FOO'<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 designator
                          and pin number }
 '(GRBX 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 }
 '(GRBX .00.12P)74LS00'#2*1: { bit 2, version 1 }
  'Y'<0>:               { first logical node }
  ;                     { no node property }
```

```
  '(GRBX .00.13P)74LS00':
    'Y'<0>:              { second logical node }
    ;                    { no node property }
  NET_NAME               { the next net }
      .
      .                  { and so on }
      .
  END.
```

HOW TO READ THE EXPANDED NET LIST

The Expanded Net List is designed to be easily read. Net descriptions are marked by the NET_NAME keyword (which 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.

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 part name 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.

8.38 THE EXPANDED PART LIST

The Expanded Part List is a part list ordered by physical designator that 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

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;
```

<list of parts>
END.

where <root drawing name> is the name of the root drawing that was compiled, <compilation time> is the time and date of the compilation of the design, and <packaging time> is the time and date of when the design was packaged, and <global design properties> is a list of design-wide properties specified in the directives section of the expansion file.

Each entry in the global property list 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 name> <part type name> : ;
<logical part list>

where PART_NAME is used to mark the beginning of a part entry. <physical part name> is the physical designator for the part. It is a string of letters and digits. <part type name> is the name of the part type for the part. It is a quoted string of characters. <logical part list> is a list of all of the logical parts that are allocated to the physical part. It has the form:

SECTION_NUMBER <section number>
<logical part name> <bit> <version> :
  <logical part properties>

where <section number> is a number indicating which section of the physical part matches the logical part. <logical part name> is the name of a logical part. It is a quoted string of characters. <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 name. <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, a semicolon is output alone.

AN EXAMPLE EXPANDED PART LIST

An example of an Expanded Part List:

```
FILE_TYPE=EXPANDEDPARTLIST;
DIRECTIVES
  ROOT_DRAWING='RISC';
  COMPILE_TIME=' COMPILATION ON THU JUN 30 11:38:02 1983';
  POST_TIME='19-OCT-1983 ';
END_DIRECTIVES;
PART_NAME
U31                    { physical part U31 }
 '74LS00':;
SECTION_NUMBER 1
 { first logical part }
 '(GRBX 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 }
 '(GRBX .00.2P)'#2*1:; { no properties }
SECTION_NUMBER 3
 { third logical part }
 '(GRBX ABD2P#2 .00.5P)':
  SIZE='2',
  TIMES='1';
SECTION_NUMBER 4
 { fourth logical part }
 '(GRBX FGH1P DD2P .00.11P)':;
PART_NAME              { the next part }
     .
     .                 { and so on }
     .
END.
```

HOW TO READ THE EXPANDED PART LIST

The Expanded Part List is designed to be easily read. Part descriptions are marked by the PART_NAME keyword (which cannot be confused with a part name since physical part names cannot contain the underscore character ('_'). The part name entry is always terminated by a semicolon (';') which comes at the end of the part type property list. The physical part name 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. The logical part name is in quotes making it easy to interpret. The logical part name 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.

Packager Physical Part Tables

8.39 INTRODUCTION

Physical part tables give you the ability to assign a property to a part that causes the Packager to invent a new part type from the basic part type. For example, a resistor may have a VALUE property attached which causes a different PART_NUMBER to be used, but otherwise the part definition would be the same as for an unselected resistor. 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 types definitions. An important use of this capability is the addition of new properties to the libraries for certain interfaces such as SCICARDS. 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 which physical part table to be used by the Packager.

8.40 A TYPICAL USE OF PHYSICAL PART TABLES

The most obvious case where physical part tables are used is with resistors and capacitors. You should be able to place a resistor into a drawing and worry about its value later. The value of the resistor should be specified with a property attached to the body so that you can easily modify it and not have to worry about alternate part names.

To achieve this, a table must be provided that relates resistance values to part numbers. One method might be to have a unique PART_NUMBER property for each resistance value. Such a table might appear as:

1K	CB1025
1.2K	CB1225
1.5K	CB1525
2.2K	CB2225

.
.

Whenever the Packager encounters a resistor, it looks up its value (specified by the VALUE property) in the table and uses the associated PART_NUMBER specified. The part lists produced would list each resistor by part number.

This method can be generalized somewhat to make it possible to associate other properties with each part instance. For example, the above table might be extended to include power dissipation, temperature coefficient, cost, reliability data, etc.

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

An example of a physical part table for 1/4 watt resistors might appears as follows:

```
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.
```

where the line numbers on the left are not actually in the file but will be used to describe the format of this table.

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 '}' and can appear anywhere a space can when used as a separator. Comments may cross line boundaries and 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 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.

8.41 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 an 'END_PART'.

PART TYPE PROPERTY LIST

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. <property name> is a standard SCALD property name which is a string of letters, digits, or '_' starting with a letter and is no longer than 16 characters.

The <property value> can be any string of characters and 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'
```

TABLE FORMAT DEFINITION

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 lists 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 follow:

```
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 ('|').

TABLE ENTRIES

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).

8.42 HOW TO USE THE PHYSICAL PART TABLES

To tell the Packager the names of the files which contain physical part tables, you must use the PART_TABLE_FILE directive. Any number of tables can be specifed 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';
```

When the Packager searches the physical part table entries, the property values on the instance must match exactly with 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 and 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, one should use '1.234K' instead of '1234', and '1MEG' instead of '1000K' or '1000000'.

Packager Cross References

8.43 INTRODUCTION

The Packager produces several cross references intended to provide information about a design that is not readily available in the drawings. These cross references include both physical and logical information and how they correspond. They are organized to provide access to the design as a whole or to a particular drawing.

This document describes the format and content of each of the cross references provided. It also describes how they are intended to be used and how the information in each is related to the others.

8.44 GENERAL PHILOSOPHY

There are two types of cross references provided: those that deal with the entire design and those that deal with a particular drawing. The term global is used to indicate a cross reference dealing with the design as a whole. The term local is used to indicate a cross reference that deals with a single drawing.

In general, global cross references are sorted by physical information. This is because this is the most common access required when looking for something in the entire design. For example, the designer may wish to know which parts are connected by a specific net, which nodes are on a specific net, what logical parts are in a specific physical part, etc.

Local cross references, on the other hand, are usually sorted by logical information. This is because they refer to a specific drawing which represents the logical design exactly but may contain only vague information about the physical design. The designer needs to know where the logical parts on a print are to be found in the physical design, what physical names were assigned to the logical signals in the drawing, what the physical pin assignments are for each logical pin, etc.

The Packager supports local and global cross references containing many different types of information. For each local cross reference, there is usually a corresponding global cross reference. The two contain basically the same information, but are sorted differently.

A description of conventions used in the cross references is located at the end of this section.

8.45 THE LOCAL PART CROSS REFERENCE

A Local Part Cross Reference is produced for each drawing in the design. It is a list of all of the logical parts (sorted by logical part) that appear in the drawing. 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
  - Pin name for each pin
  - Physical net name connected to each pin
  - Logical signal connected to each pin

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

<logical part name> <PATH property> <physical part>

<pin number> <physical net> <pin name> <logical net>

.
.
.

An entry in such a cross reference for the 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 this logical part was placed in (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 changes to summarize the common portions of the logical part and then to list each of the SIZE and/or TIMES replicated sections. For example, a 100145 with a SIZE=4 property might appear in the cross reference as:

```
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), a SIZE value specification showing how many SIZE and/or TIMES replicated parts there

are (SIZE=4). Following the first line, the common pins of the part are shown in the same format as in the 100166 example shown above:

<pin number> <physical net> <pin name> <logical net>

as, for example:

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

Following the common pins summary, each SIZE and/or TIMES replicated section of the logical part is shown. The form for each SIZE and/or TIMES replicated section is:

Section: <PATH element> <physical part name>

<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. It only lists those pins that are unique to the specified section.

8.46 GLOBAL PART CROSS REFERENCE

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

Physical part name
Part type
List of the nodes on the part with:
Pin number of the 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 name> <part type>

<pin number> <phys net> <log net> <PATH> <drawing>

.
.
.

An entry in such a cross reference for the 100145 might

appear as follows:

```
U1   100145
 1    READADRB2     READ ADR B<2>     26P#3   GRBX MJP.LOGIC.1.1
 2    READADRB1     READ ADR B<1>     26P#3   GRBX MJP.LOGIC.1.1
 3    READADRB0     READ ADR B<0>     26P#3   GRBX MJP.LOGIC.1.1
 4    OPB3          OP B<3>           26P#3   GRBX MJP.LOGIC.1.1
 5    OPB2          OP B<2>           26P#2   GRBX MJP.LOGIC.1.1
 8    OPB1          OP B<1>           26P#1   GRBX MJP.LOGIC.1.1
 9    OPB0          OP B<0>           26P     GRBX MJP.LOGIC.1.1
 10   RESULT0       RESULT<0>         26P     GRBX MJP.LOGIC.1.1
 11   RESULT1       RESULT<1>         26P#1   GRBX MJP.LOGIC.1.1
 12   RESULT2       RESULT<2>         26P#2   GRBX MJP.LOGIC.1.1
 13   RESULT3       RESULT<3>         26P#3   GRBX MJP.LOGIC.1.1
 14   C2L           -C2               26P#3   GRBX MJP.LOGIC.1.1
 15   C2L           -C2               26P#3   GRBX MJP.LOGIC.1.1
 16   C1L           -C1               26P#3   GRBX MJP.LOGIC.1.1
 17   C1L           -C1               26P#3   GRBX MJP.LOGIC.1.1
 19   MR            MR                26P#3   GRBX MJP.LOGIC.1.1
 20   WRITEADRAB0   WRITE ADR AB<0>   26P#3   GRBX MJP.LOGIC.1.1
 21   WRITEADRAB1   WRITE ADR AB<1>   26P#3   GRBX MJP.LOGIC.1.1
 22   WRITEADRAB2   WRITE ADR AB<2>   26P#3   GRBX MJP.LOGIC.1.1
 23   WRITEADRAB3   WRITE ADR AB<3>   26P#3   GRBX MJP.LOGIC.1.1
 24   READADRB3     READ ADR B<3>     26P#3   GRBX MJP.LOGIC.1.1
```

The first line of the entry shows the <physical part name> which, in the above example, is U1. Following that is the part's part type, 100145. The following lines show the pins of the part in order. For each pin, the pin number, physical net name, and logical net name are given. The last two elements of the line show the instance of the logical part that corresponds to the pin. The logical part is described by giving its path element (PATH property of the logical part as shown on the drawing and the value of the SIZE and/or TIMEs replication index) and the drawing on which the logical part appears.

The first entry in the pin list above appears as:

```
 1    READADRB2     READ ADR B<2>     26P#3   GRBX MJP.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 (that is, the logical part allocated to this physical part with a pin corresponding to this particular pin on the physical part) has a PATH property of 26P and has the SIZE replication index of 3. It appears in the drawing GRBX MJP.LOGIC.1.1.

8.47 GLOBAL SIGNAL CROSS REFERENCE

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
Input load on the net
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

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

<physical net name> <net loading> <logical signal>

<phys part> <pin #> <pin name> <part type>
<path element>
<drawing name>
.
.
.

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    GRBX MJP.LOGIC.1.1
                              26P#2    GRBX MJP.LOGIC.1.1
                              26P#1    GRBX MJP.LOGIC.1.1
                              26P      GRBX MJP.LOGIC.1.1
  U2   24   Q<26>   100150   33P#26   GRBX MJP.LOGIC.1.1
```

The first line of the above entry shows the <physical net name> (READADRB2). Following the net name is the total load on the net presented by all of the inputs on the net. Both the 0-state (3.0) and the 1-state (-3.0) loading is 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 entry in the node list has the form:

<physical part name> <pin #> <pin name> <part type>
<logical part name>

In the above example, the <physical part name> is U1, the <pin #> is 1, the <pin name> is AR<2>, and the <part type>

is 100145. The <logical part name> is given by a PATH element (26P#3) and the name of the drawing in which the part appears (GRBX MJP.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. The following node list for U1 would be equivalent:

```
U1   1   AR<2>   100145   26P#3   GRBX MJP.LOGIC.1.1
U1   1   AR<2>   100145   26P#2   GRBX MJP.LOGIC.1.1
U1   1   AR<2>   100145   26P#1   GRBX MJP.LOGIC.1.1
U1   1   AR<2>   100145   26P     GRBX MJP.LOGIC.1.1
```

8.48 CONTROLLING CROSS REFERENCE GENERATION

The cross references are generated by the Packager under the direction of the Packager 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, the directive

```
OUTPUT CROSSREFERENCES;
```

is used. This causes ALL of the cross references to be generated. Each cross reference may be individually selected as well. To cause all of the cross references to be generated by individually selecting them, the directives

```
OUTPUT LOCALPARTXREF;
OUTPUT GLOBALPARTXREF;
OUTPUT GLOBALSIGNALXREF;
```

are used or, equivalently, the single directive

```
OUTPUT LOCALPARTXREF,GLOBALPARTXREF,GLOBALSIGNALXREF;
```

can be used.

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 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 they are specified in the OUTPUT directives. The order is:

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

If you do not wish to manually break the PSTXREF file into separate cross references, they may be individually generated by selecting only one cross reference output at a time and re-running the Packager.

The cross references are ALL output assuming at least 132 characters are permitted in a line. There is no provision for you to specify the width of the output file.

8.49 HOW TO USE THE CROSS REFERENCES

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

LOCAL PART CROSS REFERENCES

The Local Part Cross Reference 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 parts that are found in the drawing. It identifies a part by giving its name and the PATH property attached to it. Given a part in a drawing, the designer can easily find the corresponding entry in this cross reference since it is ordered by part name. If there is more than one instance of a particular part within the drawing, the specific 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 physical part (because of SIZE or TIMES replication), the physical part is given for each of the SIZE and TIMES replicated logical parts; each of which is listed separately.

For each pin on the part, the logical and physical signal names are given. 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 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 PART CROSS REFERENCE

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 the part appears in. To find the part, get the drawing referenced (it refers to a specific page), find a part on the drawing of the given name (the physical part's part type is the same as the name of the part in the drawing), and make sure it has the PATH property given in the PATH element (see below for a description of how PATH elements are displayed).

The physical net name can be looked up in the Global Signal Cross Reference to find out to which other parts it is connected.

GLOBAL SIGNAL CROSS REFERENCE

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.

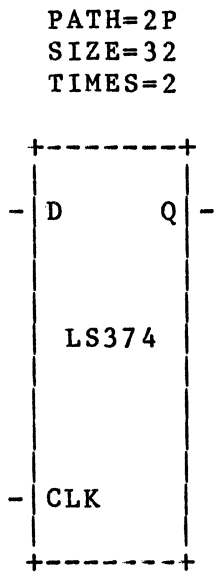
Each node on the net is listed. For each node, the physical part name and pin number are given. This makes it possible to trace a net in the physical design. To make it possible to understand what the node is, the part type and pin name are also given. To make it possible to find the node in the drawings, the logical part corresponding to the particular pin is given by specifying a PATH element and the drawing in which the logical part can be found.

8.50 CONVENTIONS USED IN CROSS REFERENCES

Descriptions of various conventions used within the cross references appear in this section.

PATH PROPERTIES AND PATH ELEMENTS

PATH information is used in several places to specify a specific 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. Take, for example, the following part:



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 (therby doubling the number of components).

The PATH property for the LS374 is just as seen in the drawing: 2P. This refers to a particular component as it appears in a drawing. A component in a drawing may refer to more than one logical component. The LS374 above represents 64 logical components. The SCALDsystem 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, following the SIZE index, separated by a ''. 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 whenever the SIZE or TIMES index value is 0, it is not output. This is done so to prevent SIZE (#0) and TIMES (0) values on parts which have no SIZE or TIMES properties.

Packager State Files

8.51 INTRODUCTION

The Packager generates and reads the part bindings (PSTPRTB), signal bindings (PSTSIGB), 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, 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 during ASSIGN PHYSICAL PARTS and ASSIGN PHYSICAL NET NAMES, and PERFORM PIN SWAPS.

8.52 LOGICAL TO PHYSICAL ASSIGNMENTS

When the Packager builds a physical design to match the logical design presented in the drawings, it assigns physical sections to logical parts, assigns physical names to logical nets, and assigns physical pins to logical nodes. Some logical parts may correspond to several physical sections (due to SIZE expansion), and some logical nets may correspond to several physical nets (due to TIMES expansion and signal versioning). The part, net and pin assignments are always performed so that the resulting physical design matches the logical design.

A small change in the logical design must result only in a corresponding small change in the physical design. This makes it possible to modify a design while physical design is in progress without requiring physical layout to be started over.

State files provide the Packager the assignments from the previous run. Those assignments which are still legal in the current run (the parts, nets, or pins they reference still exist in the logical design) are performed. Any new logical parts, nets, or pins are then assigned.

The logical changes (PSTLCHG) file contains a list of changes in the logical to physical part assignments from the last time the design was packaged. These changes include addition and deletion of logical parts and reassignment of logical parts to different physical sections. The temporary form of this file reports each logical part for which the assignment has changed, and most information about the

physical section to which it is or was assigned. No summary of net or pin changes is generated.

8.53 HOW TO USE STATE FILES FOR NORMAL OPERATION

When enabled, the Packager reads state files 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 systems, a separate disk. This ensures that a state file for one design will not be applied to a different design.

To disable the use and generation of state files, the Packager directive USE_STATE_FILES OFF; must be used since the default is ON. Because the default is ON, you must insert this directive to keep the Packager from using or generating new state files.

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.

8.54 MODIFYING THE PHYSICAL DESIGN

Since the state files direct the logical to physical assignments performed by the Packager, component allocation changes, physical part name changes, physical net name changes, and physical pin assignments made to the physical design during layout may be fed back into the Packager via the state files. Subsequent runs of the Packager will then use the new physical information for all forms of output. This facilitates documenting the final form of a design, or making changes in a design which has already been laid out.

The physical design may be modified as often as desired. Typical times for feeding back physical information might be at completion of wire-wrap prototype design and completion of PC board design.

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 the Packager Reference Manual.

8.55 STATE FILE FORMATS

The state files are intended for use only as internal files for the Packager. The formats of the part bindings, signal bindings, pin swap, and design state files may vary slightly in future releases.

PART BINDINGS FILE

The current part bindings file is structured as follows:

```
FILE_TYPE=PART_BINDINGS;
<part binding entry>
        .
        .
        .
END.
```

Each <part binding entry> is of the form:

```
<logical part name>
    <section assignment> ... <section assignment>
;
```

Each <section assignment> is of the form:

```
<size> <version> <physical part name> <pin number>
```

where <physical part name> is the name of the physical part containing the section and <pin number> is the number of the pin connected only to the section of the part matching the logical section (it is a non-common pin). The <size> is of the form:

```
# <SIZE offset>
```

where <SIZE offset> specifies the particular bit of a SIZE replicated component. Bits of the component are numbered from 0 to number of bits - 1. The <version> is of the form:

```
* <version number>
```

where <version number> specifies which version of the TIMES replicated component. Versions of a part are numbered from 0 to number of versions - 1.

An example part bindings file:

```
FILE_TYPE=PART_BINDINGS;
'(T12 .8BA12P ADC2P STL10P)'
#0*0 'U17' 2
;
'(T12 .8BA12P ADC2P STL11P)'
#0*0 'U18' 2
;
'(T12 .8BA12P ADC2P STL12P)'
#0*0 'U20' 2
;
'(T12 .8BA12P ADC2P STL13P)'
#0*0 'U25' 2
;
'(T12 .8BA12P ADC2P STL15P)'
#0*0 'U40' 2
#0*1 'U40' 3
#0*2 'U40' 4
#0*3 'U41' 2
#0*4 'U40' 5
;
'(T12 .8BA12P ADC2P STL16P)'
#0*0 'U50' 2
#0*1 'U50' 3
#0*2 'U51' 2
#0*3 'U50' 4
#0*4 'U50' 5
;
END.
```

SIGNAL BINDINGS FILE

The current signal bindings file is structured as follows:

```
FILE_TYPE=SIGNAL_BINDINGS;
<signal binding entry>
          .
          .
          .
END.
```

Each <signal binding entry> is of the form:

<logical net name> <bit> <version> <physical net name> ;

where the <logical net name> is the logical name of the net and <physical net name> is the physical name of the net. These are output as quoted strings. The <bit> is only output for vectored nets and has the form:

'<' <bit number> '>'

where <bit number> specifies which bit of the vectored net is assigned the physical name. The <version> has the form:

* <version number>

where <version number> specifies which version of the TIMES replicated net is assigned the physical name. If the <version number> is 0, the <version> is not output.

An example signal bindings file:

```
FILE_TYPE=SIGNAL_BINDINGS;
'(T12 .8BA12P ADC2P)UN$1$"STL#"$10P$A'<0>
'UN1STL10PA0';
'(T12 .8BA12P ADC2P)UN$1$"STL#"$11P$A'<0>
'UN1STL11PA0';
'UN$1$"TSN#"$10P$Y'<3>*3
'UN1TSN10PY3V3';
'UN$1$"TSN#"$10P$Y'<3>*2
'UN1TSN10PY3V2';
'UN$1$"TSN#"$10P$Y'<3>*1
'UN1TSN10PY3V1';
'UN$1$"TSN#"$10P$Y'<3>
'UN1TSN10PY3';
'-(T12 .8BA12P)PINNAME$COUT'
'PINNAMECOUTL';
END.
```

PIN SWAP FILE

The current pin swap file is structured as follows:

```
FILE_TYPE = PIN_SWAP;
<pin swap entry>
        .
        .
        .
END.
```

Each <pin swap entry> is of the form:

<logical node name> : <new pin number> ;

where <logical node name> is the logical name for a node and <new pin number> is the new pin number for the node. The colon (':') is used to separate the end of the <logical node name> from the start of the <new pin number>.

The <logical node name> is of the form:

```
<logical designator> <size> <version> : <pin name>
```

where <logical designator> is the name of the logical part to which the node belongs. It is a quoted string of characters. <size> is an optional field of the following form:

```
# <SIZE offset>
```

where <SIZE offset> specifies the particular bit of a SIZE replicated component. If the <SIZE offset> is 0, the <size> is not output. <version> is an optional field of the form:

```
* <version number>
```

where <version number> specifies particular version of the TIMES replicated component. If the <version number> is 0, the <version> is not output.

<pin name> is the name of the logical pin which the node corresponds and has the Valid canonical syntax as follows:

```
'<assertion><name>' <subscript>
```

where the assertion and name of the pin are enclosed by quotes. The <assertion> character is '-' and only appears for low asserted signals. The <subscript> is only present for vectored pins and can only be a single bit. The colon (':') is used to mark the end of the logical part name and the start of the pin name.

An example pin swap file:

```
FILE_TYPE = PIN_SWAP;

  { the following two entries swaps
    pins 1 and 2 of a 74LS00 }

'(GRBX .00.12P)'#2*1:      { bit 2, version 1 }
'A'<0>:                    { pin name and bit }
2;                         { new pin number }
'(GRBX .00.12P)'#2*1:      { bit 2, version 1 }
'B'<0>:                    { pin name and bit }
1;                         { new pin number }
```

```
          .
          .                          { and so on }
          .
     END.
```

DESIGN STATE FILE

The current design state file is structured as follows:

```
FILE_TYPE=STATE_FILE;
ROOT_DRAWING='<root drawing name>';
TIME='<compilation time>';
END.
```

where <root drawing name> is the name of the root drawing that was compiled and <compilation time> is the date and time that the design was last compiled.

8.56 FORMAT OF THE LOGICAL CHANGES FILE

The format of the logical changes file (PSTLCHG) is temporary and WILL CHANGE in future releases. The notation used in the logical changes file is similar to that of the part bindings file. Logical sections are listed as DELETED or ADDED to the design. Logical sections which existed in both the last run and the current run of the Packager, but which no longer fit into the same physical sections are listed as REASSIGNED from their old physical sections and ADDED to their new physical sections.

An example logical changes list:

```
TEMPORARY LOGICAL CHANGES LIST - 1  25-FEB-1983 15:51:49.82
LOGICAL PARTS DELETED FROM DESIGN:
  (T12 .8BA12P ADC2P STL16P) STL#;
    Reassigned: #0*2 U40 5
  (T12 .8BA12P ADC2P STL21P) STL#;
       Deleted: #0*4 U551 2
       Deleted: #0*4 U650 5

LOGICAL PARTS ADDED TO DESIGN:
  (T12 .8BA12P ADC2P STL15P) STL#;
         Added: #0*4 U40 5
  (T12 .8BA12P ADC2P STL16P) STL#;
         Added: #0*2 U51 2
         Added: #0*3 U50 4
         Added: #0*4 U50 5
END LOGICAL CHANGES LIST
```

Packager Glossary of Terms

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

8.57 TERMS

LOGICAL PART
A section of a physical part. It may have SIZE and TIMES properties attached that are used by the Packager to generate several logical parts (sections).

LOGICAL PART TYPE
The logical part type is the name of the logical part. This is always the same as the physical part type (which is the name of the physical part). The logical part is assigned in the libraries as the PRIMITIVE property attached to the .PART drawing. If the PRIMITIVE property is not present, the logical part type is the same as the drawing name.

LOGICAL PART DESIGNATOR
The name given to each instance of a logical part. The name is assigned by the Compiler and consists of the path name for the part and the logical part type. The path is augmented by the Packager when replicating parts with SIZE properties or creating new versions because of TIMES properties.

LOGICAL PIN NAMES
The name given a pin of a SCALD body.

LOGICAL PIN DESIGNATOR
Consists of a logical part designator and a logical pin name separated by a space.

PHYSICAL PART
Something understood by the physical design system. For instance, a 74LS00 might be known by some layout program and this would be a physical part. In the case of a gate array design, a physical part would be a component of the array. In general, a physical part is something that a physical design system is going to try to hook up. A physical part is normally associated with a package and not a section of a package. A NAND gate within a 74LS00 is not a physical part but the 74LS00 is.

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 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 may be assigned by the designer or is automatically assigned by the Packager.

PHYSICAL PIN NAME
The name given a pin on a physical part. This name is, by convention, a number or an identifier (not more than 16 characters) and is specified by the PIN_NUMBER property on the library component describing the physical part type.

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

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.

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.