CHAPTER 5

COMPILER

An Overview of the Compiler

5.1 INTRODUCTION

This section is intended to introduce the user of the SCALDsystem to the SCALD Compiler. It describes the purpose of the Compiler, its inputs and outputs, the tasks it performs, and how to interpret the messages it produces. Several other sections are mentioned that describe in detail topics summarized here.

The SCALDsystem user creates drawings with the Graphics Editor. These drawings may be "flat", this is, containing only "real" components (references to real devices such as TTL or ECL parts,) or they may be hierarchical, containing references to "abstract" drawings that refer to some other collection of components that may, in turn, be defined in terms of other components, etc.

Physical design systems, in general, expect designs that are "flat"; they are designed to connect (layout) circuits made up of physical components and cannot handle abstractions. One of the major tasks of the Compiler is to remove the abstractions from the design replacing them with physically realizable equivalents.

The Compiler also has the task of converting the graphical description of the design into a form interpretable by the rest of the SCALDsystem analysis programs and to provide a standard interface to the user's physical design systems.

The SCALDsystem process starts with the creation of drawings with the Graphics Editor. The Compiler reads the drawings, checks for errors, and outputs a description of the entire design with all abstractions removed. DIAL reads this description and outputs a net list for the user's physical design system. This net list is the only interface between the SCALDsystem and the user's physical design system.

5.2 THE COMPILATION PROCESS

The Compiler has several phases each of which is described below. The Compiler is run on all machines with the command COMPILE.

COMPILER DIRECTIVES

The Compiler directives are used to direct the compilation process. There are two directives that must always be specified. The ROOT_DRAWING directive specifies the name of the drawing that is used as the start of the compilation. The DIRECTORY directive lists the SCALD directories and libraries to be used in the compilation. These directories list the names of all the drawings that might be used in the design. The Compiler directives are read from a file created by the user. This file can be easily created with a text editor.

PROPERTY ATTRIBUTES

Properties can be given attributes that describe how the property is to be interpreted by the Compiler. These attributes include specification if the contexts in which the property is to be inherited, whether it is a parameter and its type, and whether it has special characteristics when used in a signal name. These attributes are read from a file. The Compiler always reads a Valid supplied standard property file which can be found on the VAX in SYS$SCALD:PROPERTY.DAT, on the S-32 in /u0/scald/property.dat, and on the 370 in PROPERTY DATA C. You may add additional property attributes in a file specified with the PROPERTY_FILE Compiler directive.

TEXT MACROS

Special text macros can be declared that are globally known and reserved throughout the compilation. These text macros are used, for instance, to support the short form of the Timing Verifier properties in signal names. Once a text macro has been declared as reserved, another text macro of the same name is forbidden. The Compiler always reads a Valid supplied standard text macro file which can be found on the VAX in SYS$SCALD:TEXTMACRO.DAT, on the S-32 in /u0/scald/textmacro.dat, and on the 370 in TEXTMACR DATA C. You may add additional text macros in a file specified with the TEXT_MACRO_FILE Compiler directive.

ROOT DRAWING

Once the Compiler has been initialized (the above steps), it reads in the specified root drawing; the drawing which is to be used as the start of the compilation.

DRAWING PROCESSING

The drawings of the design are processed. The Compiler reads each drawing used in the design and checks the properties and signals for errors. The compilation proceeds by first processing the signals and properties attached to each body used in the drawing. Second, the drawing associated with each body is read and the process is repeated. Finally, a drawing is reached that contains no bodies (that is, it is not defined in terms of other drawings) and the process stops. The result is the creation of a "tree". The "root" of the tree is the drawing specified (with the ROOT_DRAWING directive) and the lowest level drawings (the bottom of the tree) form a "flat expansion" of the design. Each of these bottom level drawings are "primitives" in that they are not defined in terms of other drawings; they are complete in themselves.

OUTPUT OF THE DESIGN

The output expansion file is generated. At this time, the Compiler selects base signal names for all of the signals in the design.

OUTPUT OF THE SIGNAL SYNONYMS

The synonyms file is generated. For each signal encountered in each page of each drawing, the name of the base signal as it appears in the design is Generated.

ERROR HELP

Documentation of the error messages encountered during the compilation are output if enabled with the ERROR_HELP Compiler directive. This documentation is intended to help you to understand and correct the errors.

5.3 A SUMMARY OF THE FILES INPUT BY THE COMPILER

The following files are read by the Compiler. A short description of each is intended as a reminder; they are described fully elsewhere. The form of each file as well as how the file is created is described.

COMPILER DIRECTIVES
: This is a text file created by the user containing Compiler directives used to direct the compilation process. See the Compiler Directives section for details.

CONFIGURATION
: This file contains the signal syntax specification for the user's site. This file is not modifiable by the user; only by Valid Systems Engineers.

MASTER LIBRARIES
: This is a file that describes the name and location of the libraries in the system. For example, an entry might describe the LSTTL library as being in the /u0/lib/lsttl/lsttl.lib SCALD directory. The Compiler always reads the system-wide master library file (supplied by Valid) and will read additional master library files supplied by the user (see the MASTER_LIBRARY directive.)

PROPERTY ATTRIBUTES
: This is a text file containing attribute assignments for properties used in the design. These assignments are automatically read from standard Valid property attributes file and you may specify an additional file as well. See the Property section for details.

RESERVED TEXT MACROS
: This is a text file containing definitions of globally known reserved text macros. These definitions are automatically read from Valid standard text macro file and you may specify an additional file as well. See the Text Macro section for details.

CMPERRORS.MEM
: This file contains error message documentation. It is read by the Compiler to output error documentation at the end of the compilation. See the Compiler Error Message section for details.

DIRECTORIES
: These files are specified with the DIRECTORY Compiler directive. They are used by the Compiler to find the drawings used in the design. Directories are only created by the Graphics Editor. See the SCALD Directory section for details.

DRAWINGS
: These files contain descriptions of drawings; one drawing per file. The Compiler reads these as needed. The ROOT_DRAWING is always read and all other drawings encountered during the compilation are first found in the directories and then read.

5.4 A SUMMARY OF THE FILES OUTPUT BY THE COMPILER

The following files are written by the Compiler. A short description of each is intended as a reminder; they are described fully elsewhere. The form of each file as well as how the file is created is described.

MONITOR
: This file is the standard output file. It shows a summary of the execution of the Compiler listing each phase of the compilation and the time each phase took. A summary of the errors, oversights, and warnings is given at the end. This file is directed to the standard output device; the terminal or a line printer. This file cannot be suppressed.

CMPLOG
: This file contains all error, warning, and oversight messages produced during the compilation (as well as those warning and oversight messages that were suppressed). Any internal errors (error 187) detected are also printed here. Execution statistics are output as well. This file is intended to provide Valid personnel with information needed when fixing problems with the programs. It should be saved whenever an

ASSERTION FAILURE error message is encountered. The file cannot be suppressed.

LIST
: This file contains the execution summary and all error messages. It is an expanded form of the information written to MONITOR. A message is printed for each drawing processed. This file can be suppressed with the OUTPUT Compiler directive.

EXPAND
: This file contains the "flat expansion" for the design. It is a list of all the "real" components and how they are connected. This file can be suppressed with the OUTPUT Compiler directive, except with separate compilation when it is always produced.

SYNONYM
: This file contains lists of signal synonyms (or aliases) detected during the compilation. It is used by other SCALDsystem programs to refer to all names for a given net. This file can be suppressed with the OUTPUT Compiler directive, except when there is separate compilation in which case it is always produced.

CHIPS
: This file contains chip definitions used by the SCALD Packager. The process for generating this file is described in the Generating Physical Libraries documentation. This file can be suppressed with the OUTPUT directive.

5.5 ERRORS DETECTED BY THE COMPILER

As the Compiler processes the design, it detects several types of errors. The Compiler makes an attempt to "correct" errors as they are detected to reduce the number of error messages that would be produced as the error propagates through the design. The Compiler attempts to have each error cause one error message to be printed.

ERROR MESSAGE TYPES

The Compiler groups error conditions into three types: warnings, oversights, and errors. A warning condition can usually be ignored. An oversight is more serious, but the

design as produced by the Compiler is probably okay for some types of analysis. An error should be corrected before continuing. See the Compiler Error Messages section for a complete discussion of error messages.

ERRORS DETECTED

The Compiler detects many different types of errors. The general types are described below.

SYNTAX ERRORS

These errors are detected as the Compiler reads various files such as the Compiler directives, reserved text macros, or property attributes files. The syntax (form) of the statements within these files is checked for correctness. When the form is incorrect, a syntax error is generated.

Syntax errors can also appear in signal names. In this case, the form of a signal name is in error. The Compiler guesses at the form the signal is supposed to take.

DRAWING ERRORS

These errors are detected as the Compiler reads a drawing. They include errors such as: undefined drawing name (missing from directories), selection expression errors (such as more than one selection evaluates true), incompatible drawing types (such as .PRIM and .PART), missing standard properties (such as ABBREV and PATH), and invalid file type (the file that is supposed to contain the drawing definition has the wrong type).

SIGNAL ERRORS

A large number of errors may be detected in the processing of signals. Signal errors are detected as the Compiler processes the bodies found in a drawing. For each body, the pin names and signal names are read and checked.

The checks performed on pin names and signals connected to them are:

1. Check that the pin name is not a constant, a concatenated or replicated signal, has consistent properties, and is unique (that is, different from all other pin names on the body).

2. Make sure the signal name has the correct scope, that it exists, and is used correctly.

3. Check that the signal name's width matches the pin name's width.

4. Check that the signal name's assertion matches the pin's bubble state.

5. Check that the pin properties do not conflict with the signal or its properties.

Whenever a signal error is detected, the Compiler prints the drawing currently being read, the body to which the signal in error is connected, the name of the pin on that body, and the signal itself. The Compiler tries its best to repair the incorrect signal. It recovers by concatenating an NC signal (for width errors), replacing it with an NC (for catastrophic errors), etc. The attempt is made to reduce the impact of the error on the rest of the compilation.

5.6 HOW TO COMPILE FOR TIMING VERIFICATION OR SIMULATION

The Compiler always produces output intended for some other SCALDsystem program or for transfer to another CAD system (using DIAL). By default, the Compiler compiles to physical parts. This can be used by the Packager or by DIAL. The output of the Compiler intended for the Packager consists of a description of the design at the physical part level; that is, the design is compiled until only "real" parts are left (such as TTL or ECL components). The Packager has information about these components (their pin numbers, loading characteristics, etc.) that it uses to package the design.

The Timing Verifier, on the other hand, understands a different collection of components. These components are called Timing Verifier Primitives. A design must be expressed in terms of these parts if the Timing Verifier is to understand it. The Valid supplied library components have Timing Verifier models as well as physical models. The designer can compile the design for timing verification by informing the Compiler with the COMPILE directive in the Compiler directives file. The compile directive is used to specify the drawing extension to be used in place of a .PART. Compilation for the Timing Verifier uses drawings with the .TIME extension. The Compiler directive to use is COMPILE TIME.

Compilation for the Simulator is accomplished in a manner similar to that for Timing Verification. The drawing extension to be used is .SIM. The Compiler directive is COMPILE SIM.

For a more complete description of the use of SCALD directories and drawing extensions, see the SCALD Directory section.

SCALD Directories

5.7 INTRODUCTION

In the SCALDsystem, drawings are given names invented by the designer and placed in unique physical files. There are no restrictions placed on the form SCALD drawing names may take but the operating system used to support the SCALDsystem (both in UNIX and on the user's host) has a restrictive file naming convention. For this reason, it is not possible to use the SCALD drawing name as the physical file name.

The solution is a special directory, called the SCALD directory, which maps SCALD drawing names to physical file names created automatically by the Graphics Editor. These SCALD directories serve two purposes. First, they permit the designer to refer to the drawings by the drawing name and to ignore system names. Second, the directories permit the designer to ignore system specific file naming conventions since these are handled automatically. The SCALD directories make the file system of the supporting machines (in UNIX or the host) transparent to the designer. Since many hardware designers may have little knowledge of the computer systems being used for CAD, hiding detailed operational knowledge is a big advantage.

5.8 THE SCALD DIRECTORY TYPES

SCALD directories are given special "types" that identify the function of the directory. There are three standard directory types: LOGIC, TIME, and SIM. Each of these is described below.

LOGIC

A LOGIC directory (type = LOGIC_DIR) contains drawings created by the designer. This is the default directory type and it is unlikely the designer will ever have to work in another type of directory. Drawings with any type can be placed in a LOGIC directory (see the section on drawing types). The standard types are BODY, LOGIC, TIME, SIM, PART, and PRIM. When a new directory is created with the Graphics Editor, it is a LOGIC directory.

TIME

A TIME directory (type = TIME_DIR) contains drawings that describe Timing Verifier primitives (those special parts that are understood by the Timing Verifier and used to construct timing models). A TIME directory may contain only drawings with the TIME or PRIM types. Timing Verifier primitives are defined by drawings with the .PRIM type. These primitives are predefined within the Timing Verifier and should not be changed.

SIM

A SIM directory (type = SIM_DIR) contains drawings that describe Logic Simulator primitives (those special parts are that understood by the Logic Simulator and used to construct simulation models). A SIM directory may contain only drawings with the SIM or PRIM types. Logic Simulator primitives are defined by drawings with the .PRIM type. These primitives are predefined within the Logic Simulator and should not be changed.

The designer may create special directory types. If, for example, a special purpose simulator is available for which a special set of primitives is needed, a directory containing these primitives can be created and given the name (for example) MYSIM_DIR. Within this directory, drawings with the MYSIM and PRIM types are permitted.

There are two directory types that are forbidden: PRIM_DIR and PART_DIR. The types PRIM and PART have special meanings in the SCALDsystem and directories of these types are meaningless. The Compiler will produce an error message if it encounters directories of these types.

Within the libraries supplied by Valid are a number of directories with special types. Some examples are SPICE_DIR, MCLDL_DIR, LOGCAP_DIR, and TEGAS5_DIR.

5.9 DRAWING TYPES

A SCALD drawing is given a type name that identifies the drawing type. Several drawings may have the same name but different types. There are six standard drawing types: BODY, LOGIC, TIME, SIM, PART, and PRIM. For each BODY drawing there must be a corresponding LOGIC drawing and, occasionally, a TIME and SIM drawing as well. Each of these drawing types is described below.

BODY

A BODY is the symbolic representation for a drawing. It is used to refer to a collection of logic without the need to include that logic in a drawing.

LOGIC

A LOGIC drawing is the standard type drawing created by the designer. It is used to define a circuit made up of parts (such as TTL or CMOS) defined in libraries or in directories created by the designer. A LOGIC drawing may contain bodies defined in LOGIC directories only (bodies defined in TIME or SIM directories may not be added to a LOGIC drawing).

TIME

A TIME drawing is used to define a timing model for some other part. A TIME drawing may contain bodies defined in a LOGIC directory or a TIME directory. If a body from a LOGIC directory is used, that body should refer only to drawings that use timing verifier primitives; that is, no drawings referring to Logic Simulator primitives should appear. The Compiler will produce an error message if this is not true.

SIM

A SIM drawing is used to define a simulation model for some other part. A SIM drawing may contain bodies defined in a LOGIC directory or a SIM directory. If a body from a LOGIC directory is used, that body should refer only to drawings that use Logic Simulator primitives; that is, no drawings referring to Timing Verifier primitives should appear. The Compiler will produce an error message if this is not true.

PART

A PART drawing is used to define a part; usually within a library of parts. The part drawing contains physical information such as power, cost, size, weight, etc. that is of some use to the designer's physical design system. The .PART drawing itself contains no logic; it is a place holder for physical information. The presence of this drawing informs the Compiler that the corresponding component is a physical part.

PRIM

PART and PRIM are the same. For identification, use .PART for real parts and .PRIM for simulator primitives. To the compiler however, they are identical. The Compiler simply ignores .PRIM and .PART drawings that are not uniquely defined if a SCALD Directory of the correct type for the compilation being performed. For example, when compiling for TIME, only .PRIM (or .PART) drawings in a TIMEDIR will be used.

Other drawing types are possible. In the previous section, the MYSIM directory was introduced to contain primitives for a special purpose simulator. Drawings that contain MYSIM primitives are given the MYSIM type (exactly as drawings containing TIME primitives are given the TIME type).

5.10 AN EXAMPLE LIBRARY PART CONSTRUCTION

The following example is included to demonstrate the use of drawing types and directories.

Let us assume we wish to make a TTL library and to start with the LS00 part. The following steps should be followed (the order is not crucial in most cases, but the order used is the most obvious).

1. Create the TTL library directory. Create the directory file file TTL.LIB with the Graphics Editor (the file name is not important; choose whatever you like. Presumably, you'll want a name that refers to TTL and library.

2. Create the shape for the LS00. Edit LS00.BODY and draw the shape. Make sure that all of the pins are given pin names. Save the drawing as LS00.BODY.1.1 (which means drawing with name "LS00", body version 1, and page 1).

3. Add the LS00 to the TTL LIBRARY drawing. This drawing should contain one example of each of the parts in the library.

4. Physical information should be attached to the LS00 within the TTL LIBRARY drawing. This information should not be placed within the LS00.BODY drawing. If placed there, the information will appear in every drawing and expansion file used. This makes

these files unnecessarily large. Properties are attached to the pins that describe the pin number, sections (if the part has sections), input and output loading, etc. Properties are attached to the body that describe the power and ground pins, power, size, cost, inventory part numbers, reliability, or what have you.

5. Create the part description. Edit LS00.PART and attach an ABBREV property to a DRAWING body there. Also add a DEFINE body. Save the drawing as LS00.PART.1.1.

6. Create the timing model. Edit LS00.TIME and add Timing Verifier primitives (from the TIME library) to create a timing model. Save the drawing as LS00.TIME.1.1.

7. Create the Logic Simulator model. Edit LS00.SIM and add Simulator primitives (from the SIM library) to create a Simulator model. Save the drawing as LS00.SIM.1.1.

8. Compile the TTL LIBRARY drawing. Make sure the Compiler directive OUTPUT CHIPS; is used. When done, rename the CHIPS.DAT file to TTL.PRT. This is the TTL library chips file needed by the Packager and DIAL. It contains all of the physical information about the TTL parts. This should be done only after all parts have been entered. It is not necessary to do for each separate part.

The TTL directory TTL.LIB should have the following entries:

```
LS00.BODY.1.1
LS00.PART.1.1
LS00.TIME.1.1
LS00.SIM.1.1
TTL LIBRARY.LOGIC.1.1
```

which form a complete library entry for the LS00 part. Repeat for every TTL component and you have a complete TTL library! But step eight only has to be done once after steps one through seven have been completed for all components.

5.11 WHAT DOES THE COMPILER DO WITH ALL OF THIS?

The drawing types are used by the Compiler to determine which drawings are to be used during the compilation. Each compilation is done with a destination in mind: the Timing Verifier, the Simulator, the Packager, etc. The COMPILE directive is used to specify the intended compilation destination. For instance, to compile for timing verification, the directive:

```
COMPILE TIME;
```

is used. This causes the Compiler to ignore all directories that are not LOGIC or TIME directories. The Compiler ignores .PRIM and .PART drawings that are not defined in a SCALD Directory of the correct type for the compilation being performed. PRIM (and PART) drawings are read only within the TIMEDIR directories.

Similarly, if the directive COMPILE SIM; is used, the Compiler only reads in LOGIC and SIM directories and drawings with the types LOGIC, SIM, and PRIM. The output of this compile is sent to the Simulator.

When compilation for the Packager (and from there to the user's physical design system) is intended, the directive COMPILE LOGIC; is used (if no COMPILE directive appears, the Compiler assumes COMPILE LOGIC;). The Compiler reads only LOGIC directories and drawings with the types LOGIC and PART (or PRIM). The output is sent to the Packager and from there to the physical design system.

5.12 A SUMMARY OF THE ABOVE

The information presented above is summarized below. There are four rules to be followed for using drawing types and SCALD directories.

Directories

LOGIC directories can contain drawings with any types.

xxx directories can contain drawings with the xxx type or the PRIM (or PART) type only.

The directories PRIM_DIR and PART_DIR are illegal.

Types

A drawing with the xxx type can contain bodies from a LOGIC directory or an xxx directory only.

5.13 SUMMARY OF WHAT THE COMPILER DOES

The response of the Compiler to the COMPILE directive is summarized below.

COMPILE LOGIC;

The Compiler reads all LOGIC directories. Within these directories, those drawings with the PART (or PRIM) and LOGIC types are read. All others are ignored.

COMPILE xxx;

The Compiler reads all LOGIC directories and xxx directories. Within these directories, those drawings with the xxx or LOGIC types are read. Within xxx Directories, those drawings with the PRIM (or PART) type are read. All others are ignored.

Compiler Directives Summary

5.14 INTRODUCTION

The Compiler directives are used to direct the compilation process. Each directive is used to inform the Compiler about how to compile the design, control error checking and reporting, or to select Compiler outputs. The directives are placed in a text file and given to the Compiler. Each of the directives is described below. The Compiler directives and their parameters are not case sensitive. An example Compiler directives file is located at the end of this section.

Directives are not case sensitive and neither are their arguments. The only exception is file names (in UNIX only). The Compiler preserves the case of file names because the UNIX operating system is case sensitive.

Except for a few exceptions, no directive may appear in the directives file more than once. The Compiler flags each repeat appearance of a directive with an error message. The directives that may be used more than once in the directives file are:

DIRECTORY
FILTER_PROPERTY
LIBRARY
MASTER_LIBRARY
OUTPUT
PASS_PROPERTY
PRIMITIVE
PROPERTY_FILE
REPORT
SUPPRESS
TEXT_MACRO_FILE

5.15 COMPILER DIRECTIVES

BUBBLE_CHECK

Used to control whether bubble checking is performed. When the Compiler processes a signal connected to a pin, it checks to make sure that the assertion of the signal matches the bubble state of the pin. That is, a signal that asserts low can only be attached to a pin that has a bubble and a signal that asserts high may only be connected to a pin with no bubble. There may be times when bubble checking is a nuisance (as, for instance, early in a design, or when some other

problem is being looked for, or when the designer doesn't wish to follow bubble conventions). This directive is used as follows:

BUBBLE_CHECK ON;	check all signals and pins for bubble violations.
BUBBLE_CHECK OFF;	don't perform any bubble checks.

If unspecified, the Compiler assumes ON.

COMPILE

Used to specify the type of compilation to be performed. The Compiler produces output used by the Timing Verifier, the Logic Simulator, and the Packager. Each of these understands a different set of parts: the Timing Verifier only recognizes timing primitives, the Logic Simulator only recognizes simulator primitives, and the Packager only understands "real" parts. The COMPILE directive is used to inform the Compiler which primitives to output. This directive accepts a single parameter that specifies the directory type in which primitive components are found. For example,

```
COMPILE LOGIC;
```

is used to compile for the Packager. The Compiler reads all SCALD directories with the LOGIC_DIR type and outputs as primitives those components with the .part (or .prim) drawing type. A compilation for the Timing Verifier is specified as:

```
COMPILE TIME;
```

The Compiler only outputs components with the .prim (or .part) type found in TIME_DIR directories (the Timing Verifier primitives library). The rule for the COMPILE directive is as follows:

COMPILE <x>; causes the Compiler to read all drawings with the .LOGIC, .prim (or .parts), and .<x> extensions from directories of the LOGIC_DIR and <x>_DIR types only.

If the COMPILE directive is unspecified (either in the directives file or on the command line), the Compiler assumes LOGIC (compilation for the Packager).

CONST_BUBBLE_CHECK

Used to control whether bubble checking is performed on constants or not. This is a similar function to BUBBLE_CHECK but applies only to constants. See the description of the BUBBLE_CHECK directive (above) for details about bubble checking. It should be noted that the signal 1* is the same as the signal 0 (except for the assertion) since 0 is the complement of 1. This directive is used as follows:

CONST_BUBBLE_CHK ON;	check all constants and pins for bubble violations.
CONST_BUBBLE_CHK OFF;	don't perform any bubble checks.

If unspecified, the Compiler assumes OFF.

DIRECTORY

Used to specify the names of the directories where the drawings may be found. Directories are used to map the drawing names to system file names. Directories can be of many types with LOGIC_DIR the standard. Library directories have the types TIME_DIR (Timing Verifier primitives) and SIM_DIR (Logic Simulator primitives). At least one directory must be specified (the one containing the root drawing). The directories are specified in a list with each file name appearing in quotes. For example:

```
DIRECTORY 'SPECIAL.LIB', 'USER.WRK';
```

Many directory directives are permitted, each specifying one or more directory files.

A directory file name can be fully rooted. The Compiler will user the path name in the directory file name to determine where the files in the directory can be found. If no directories are specified, an error is generated and the compilation is aborted.

ERROR_HELP

Used to control the printing of error message documentation at the end of the compilation. When enabled, the Compiler prints a short description of each error that was detected during the compilation. This description explains what the error means (in the several contexts in which it may appear) and gives some suggested fixes. The documentation is printed to the main Compiler list file (controlled by the LIST option of the OUTPUT directive).

ERROR_HELP ON;	Print documentation for all occurring errors.
ERROR_HELP OFF;	Do not print any error documentation.

If unspecified, the Compiler outputs error documentation.

FILTER_PROPERTY

Used to control whether specific properties appear in the Compiler's output files. The FILTER_PROPERTY directive takes a list of property names. For example:

```
FILTER_PROPERTY FOO, GRBX, BAR;
```

prevents the properties FOO, GRBX, and BAR from appearing in the Compiler's expansion file. The FILTER_PROPERTY directive will supercede any attributes in the property attributes files. In this manner a user may cause a property to be suppressed that is normally output. See also the PASS_PROPERTY directive.

LIBRARY

Used to specify one of the libraries. This command is similar to the library command in the Graphics Editor. It frees the user from needing to know the location of the libraries being used. The user can refer to the library by name and the Compiler will find it. The form of the Library directive is:

```
LIBRARY <library name list>;
```

where <library name list> is a list of one or more library names separated by commas. The library names must be in quotes unless they are identifiers (start with a letter and consist of only letters, digits, and '_'). The library names are kept in a special file in the SCALD area and correspond to a similar file read by the Graphics Editor. The standard library can be selected with the directive:

```
LIBRARY STANDARD;

        OR

LIBRARY 'STANDARD';
```

Any number of LIBRARY directives can be used as long as a given library is not specified more than once. The standard libraries (supplied with every machine) that can be referenced are:

```
STANDARD - standard components
TIME     - Timing Verifier primitives
SIM      - Simulator primitives
PHANTOM  - phantom gates
```

Other libraries (ordered by the customer) are added to this list when they are installed on the system.

MASTER_LIBRARY

Used to specify the names of master library files. These files contain the names of the libraries referred to by the LIBRARY directive. Any number of files can be specified but the libraries specified in the master libraries must all be unique. The directive has the form:

```
MASTER_LIBRARY <file list> ;
```

where <file list> is a list of file names in quotes. An error is generated if the specified master library file does not exist. The Compiler always reads the standard SCALD master library file shipped as part of the Valid libraries.

MAX_ERRORS

Used to specify the maximum number of errors permitted before the Compiler gives up. The error limit can be set to 2000 as follows:

```
MAX_ERRORS 2000;
```

If unspecified, the Compiler assumes 1000.

OUTPUT

Used to control which output files are produced by the Compiler. There are several files that are always produced: MONITOR (an execution summary to the standard list device), and CMPLOG (a log of suppressed warning messages, assertion failures, and runtime statistics). There are other files that may be selectively produced. The names of these files are listed separated by commas or separate output directives may be used. For instance, the directive:

```
OUTPUT LIST, EXPAND, CHIPS;
```

is equivalent to:

```
OUTPUT LIST;
OUTPUT EXPAND;
OUTPUT CHIPS;
```

The output file specifiers are:

LIST

Causes the CMPLST file to be created. This file contains the compilation summary and all the error messages.

EXPAND

Causes the CMPEXP file to be created. This is the output file of the Compiler used as input by the Timing Verifier, Simulator, and Packager.

SYNONYM

Causes the CMPSYN file to be created. This file contains the signal synonyms found in the design. Used when aliases of signals are needed.

CHIPS

Causes the CHIPS file to be created. This file is used by the Packager. It describes the library of physical components that the Packager understands. See the Library Structure document for a description of how to create CHIPS files.

If unspecified, the Compiler produces the LIST, EXPAND, and SYNONYM files.

OVERSIGHTS

Used to control whether oversight messages are printed or not. An oversight is a type of diagnostic message more severe than a warning and less severe an error. The Compiler will produce (probably) correct output in the presence of oversights, but these conditions should be remedied by changing the drawings. The form is as follows:

```
OVERSIGHTS ON;      Print oversight messages.
OVERSIGHTS OFF;     Do not print oversights.
```

If unspecified, the Compiler assumes ON.

PASS_PROPERTY

Used to control whether a specific property appears in the Compiler's output files. The PASS_PROPERTY directive takes a list of properties like the FILTER_PROPERTY directive above. The

PASS_PROPERTY directive will supercede any FILTER attributes in the property attributes files. In this manner a user may cause a property to be output that is normally suppressed. See also the FILTER_PROPERTY directive.

PERMIT_NO_ASSERT

This directive is used to control whether the Compiler requires every signal to have an explicit high-assertion character. The signal syntax may specify a high-assertion character (such as H or +) or NULL. If NULL, all signals without a low-assertion specifier are assumed to be high-asserted. If a character is specified, the Compiler requires that character to be used on all high-asserted signals. If you wish to make the presence of the high-assertion character optional, the directive PERMIT_NO_ASSERT ON; is used. This permits you to use no assertion character for high-asserted signals. The form is as follows:

PERMIT_NO_ASSERT ON;	Do not require high-asserti characters.
PERMIT_NO_ASSERT OFF;	Require all signals to have high-assertion character.

If this directive is not specified, the Compiler requires all signals to have an explicit high-assertion character if a high-assertion character is defined.

PRIMITIVE

Used to force a drawing to be a primitive even though its drawing type is not .PRIM or .PART. This directive can be used to stop the compilation of some drawing. When a primitive drawing is encountered (whether primitive because it has the .PRIM or .PART type or because of this directive) the Compiler assumes that it is not to be compiled and does not compile any drawings referenced by it. If the drawing (or instance) is a .prim or a .part, the directive has no effect. Otherwise the result is the removal of the drawing (or instance) from the compilation. Compilation will stop with the specified output (or instance), but it will not be output unless it is a real .prim or a real .part.

A drawing is set to the primitive type as follows:

```
PRIMITIVE '<drawing name>';
```

where <drawing name> is the name of the drawing to be made primitive. Note that the drawing name MUST appear in quotes. When the Compiler encounters this drawing, it will assume it is a primitive REGARDLESS OF ITS REAL DRAWING TYPE. The above form of the directive forces ALL instances of a drawing to be primitive. It is also possible to force a specific instance of a drawing to be a primitive leaving all other instances unaffected. This form of the directive is:

```
PRIMITIVE '(<path name>)<drawing name>';
```

where <path name> is the path name of the specific instance to be forced to primitive type and <drawing name> is the name of the drawing. If the instance specified is part of a SIZE replication of the drawing (the path name will have a '#n' as part of the last element), the drawing is forced to be primitive and the SIZE replication is stopped. For instance, assume the FOO drawing with path name (A B F) is SIZE replicated by 5. The drawing instances would be:

```
(A B F)FOO
(A B F#1)FOO
(A B F#2)FOO
(A B F#3)FOO
(A B F#4)FOO
```

If the PRIMITIVE directive:

```
PRIMITIVE '(A B F#2)FOO';
```

is specified, only the following instances of the drawing FOO will be created:

```
(A B F)FOO
(A B F#1)FOO
(A B F#2)FOO
```

with (A B F#2)FOO being a primitive. See the Compiler path name documentation for a complete description of path names and drawing extension types. If this directive is unspecified, the Compiler forces no drawings to be primitive.

PROPERTY_FILE

Used to specifiy the name of a file containing property attributes. Property attributes determine how properties are interpreted by the Compiler. There are several attributes that can be assigned to a property. These attribute assignments are read from the file specified by this directive. The file ATTRIBUTE.DAT may be specified as the property attributes file as follows:

```
PROPERTY_FILE 'ATTRIBUTE.DAT';
```

Certain predefined properties are given attributes automatically. See the property documentation for a complete description of properties and attributes. The PROPERTY_FILE directive accepts a list of property attribute files. For example, the files ATTRIBUTE.DAT and STANDARD.DAT can be specified with:

```
PROPERTY_FILE 'standard.dat', 'attribute.dat';
```

PRINT_WIDTH

Used to control the width of the output listings. The width of of the output files LIST, SYNONYM, EXPANSION can be specified with this directive. The width may be be set to any value from 80 to 132 columns. The output formatter tries to produce listings that look "pleasing" in the width specified. The widths of the three files cannot be separately controlled. The width can be set to 120 as follows:

```
PRINT_WIDTH 120;
```

If unspecified, the Compiler assumes a width of 132.

REPORT

Used to control the generation of reports to the listing file. The Compiler can generate several reports. These can be individually turned on or off.

Each report is started on a new page. There are three reports: PATH_NAMES, HIERARCHY, and SUMMARY. For example:

```
REPORT HIERARCHY;
```

causes the hierarchy report to be generated. A report can be turned off by using a '-' in front of the report name. For example:

```
REPORT -PATH_NAMES;
```

turns off the path names report. All reports can be turned on with the REPORT ALL; directive and all reports can be turned off with the REPORT -ALL; directive. Each of the reports is described below.

The PATH_NAMES report consists of the path names of each drawing compiled. This report is normally produced by the Compiler. As each drawing instance is encountered, its path name, drawing name, and non-default parameters are printed. This list can be useful when interpreting error messages.

The HIERARCHY report describes the drawing hierarchy. For each drawing in the design, the hierarchical drawings used are listed. Indentation is used to show the depth of hierarchy. Plumbing bodies (MERGE, NOT, etc.) are not included in this list. Primitive parts are also omitted but the total number is summarized. The HIERARCHY report for a flat design consists of just the root drawing name and number of parts used.

The SUMMARY report summarizes the use of signals and drawings within the design. For signals, the total number of signals in the design, the number of unnamed signals, the number of local signals, the number of global signals, and the number of interface signals are reported. For drawings, the total number of drawings, the number of instances of those drawings, the number of "plumbing" drawings (MERGE, NOT, etc.), and the number of primitive drawings are reported.

If no REPORT directive is used, the Compiler assumes REPORT PATH_NAMES.

ROOT_DRAWING

Used to specify the root drawing for the compilation. The Compiler must be given the name of a drawing so that it knows where to start compiling. Any drawing in the design may be specified; the most global (if the entire design is to be compiled) or some low level drawing (if a small portion of the design is to be compiled). The drawing name must be placed in quotes. This directive must always appear since the Compiler cannot start the compilation without the name of the root drawing. To compile starting at the drawing ROOT DRAWING OF THE DESIGN, the following would be used:

```
ROOT_DRAWING 'ROOT DRAWING OF THE DESIGN';
```

Optional if the root drawing is specified in the command line.

SINGLE_DRAWING

Used to control whether the Compiler compiles a single drawing (the one specified with the ROOT_DRAWING directive) or compiles the entire design. This feature is supported to allow the separate compilation of drawings that are to be sent to other CAD tools (such as a logic simulator). It is assumed that the other CAD know how to process separately compiled drawings, i.e., the separately compiled drawings must be linked together. The SCALDsystem does not support this linking function for this directive.

When SINGLE_DRAWING is ON, the Compiler does not compile any hierarchical bodies called within the drawing. It does compile all plumbing bodies (MERGERs, NOTs, etc.) The hierarchical bodies are output as though they were primitives. Such an expansion file is converted to a "module" by a DIAL Interface. These "modules" are linked together by the destination CAD system (such as Tegas).

The AUTO_GEN=TRUE property is attached to each hierarchical body that is not compiled and is output into the expansion file. This property is used by DIAL and the Packager to automatically generate pin numbers and input/output descriptions for the component in the absence of a chips file. See the Packager documentation for a complete description of this property.

WARNING: Several restrictions must be placed on the use of the SCALD design language in order for compilations of a single drawing to work. No global signals may be used. The destination CAD system cannot handle references to global signals. No parameters may be used. Parameters are properties passed into a drawing to customize it. The most common parameter in the SCALDsystem is SIZE. Since the value of the SIZE property is NOT known, a compilation of a drawing using the SIZE property will produce an incorrect design (parameters may be pre-defined in the text macro file but, since every drawing to be compiled may need a different text macro file, this involves considerable logistical difficulties). No inheriting pin properties are permitted on hierarchical bodies. Since these need to be inherited down through the hierarchy, they cannot be properly handled when compiling a single drawing. All plumbing bodies used must have either one (or more) NWC pin or a PART_TYPE = 'PLUMBING' property on the body.

This directive is used as follows:

```
SINGLE_DRAWING ON;        compile only the root drawing.

SINGLE_DRAWING OFF;       compile the entire design.
```

If unspecified, the Compiler assumes SINGLE_DRAWING OFF.

SUPPRESS

Used to suppress specific warning and oversight messages. When information is left out of some drawings (such as definitions for X_FIRST or X_STEP or SIZE), the Compiler generates warnings or oversights to bring this fact to the attention of the designer. The designer may choose to add this information (which is suggested since it improves the level of documentation) or the specific warning or oversight message may be suppressed. For example, the warning #193 may be suppressed as follows:

```
SUPPRESS 193;
```

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

```
suppress 193, 194, 195;
```

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

TEXT_MACRO_FILE

Used to specify the name of the file containing globally known reserved text macro names. The Compiler permits the designer to specify text macro names that are known within all drawings in the design (can be used in all signal names). The Compiler makes sure that these names are reserved; i.e., no text macro of the same name may be defined (in a DEFINE body) anywhere within the design. The Timing Verifier timing assertion text macros are defined with a system-wide text macro file. The designer may specify any other text macros as well. See the Compiler text macro documentation for more details. The name of the file must be in quotes. The file TEXTMACRO.DAT may be specified as the text macro file as follows:

```
TEXT_MACRO_FILE 'TEXTMACRO.DAT';
```

The TEXT_MACRO_FILE directive accepts a list of text macro files. For example, the files MACROS.DAT and STANDARD.DAT can be specified with:

```
TEXT_MACRO_FILE 'standard.dat', 'macros.dat';
```

WARNINGS

Used to control whether the Compiler prints warning messages. Several conditions are detected by the Compiler that are not as severe as errors, but need to be brought to the attention of the designer. Rather than print an error message, the Compiler prints a warning indicating that the condition may be an error and should be checked. All warning conditions can be eliminated by adding the needed information (described in the warning message) to the drawing. All warning messages may be suppressed (though it is a good idea to add the information to the drawings - this information helps to more clearly document the design). The directive is specified as follows:

```
WARNINGS ON;  display all warning messages to
              the Compiler's list file.

WARNINGS OFF; display no warning messages.
```

If unspecified, the Compiler outputs all warning messages.

5.16 AN EXAMPLE OF A COMPILER DIRECTIVES FILE

The Compiler directives file can be created with a text editor. The Compiler does not pay any 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 (except under operating systems where filename case is important (for example, UNIX). Comments may be placed in the file if enclosed with '{' and '}'. The directives below cause the directories USERDIR.DAT, PRIMITV.DAT, and PARTS.DAT to be searched for drawings during compilation. The root drawing of the compilation is MAIN ALU BOX. Note that all Compiler directives must be separated by ';' and the file must end with an 'end.'.

```
directory 'USERDIR.WRK',        {my user macros}
          'PRIMITV.WRK',        {the primitives used in this
                                 design}
          'PARTS.WRK';          {more primitives used in the
                                 design}

root_drawing 'MAIN ALU BOX';    {name of the highest level
                                 drawing in the design}

BUBBLE_CHECK on;                {just to make sure}
MAX_ERRORS   1000;              {I'm careless}

PRINT_WIDTH 132;                {for the line printer}

error_help on; report all;      {multiple directives per line
                                 are fine}

end.                            {this marks the end of the
                                 file}
```

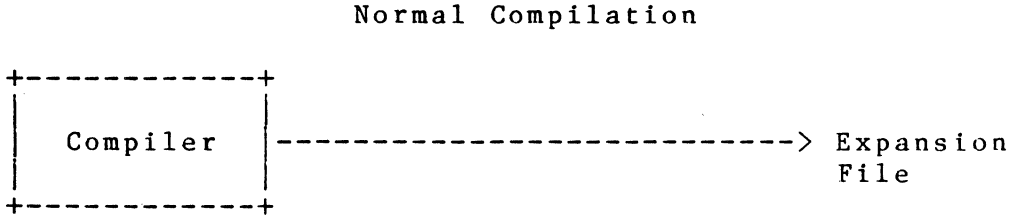
Separate Compilation

5.17 INTRODUCTION

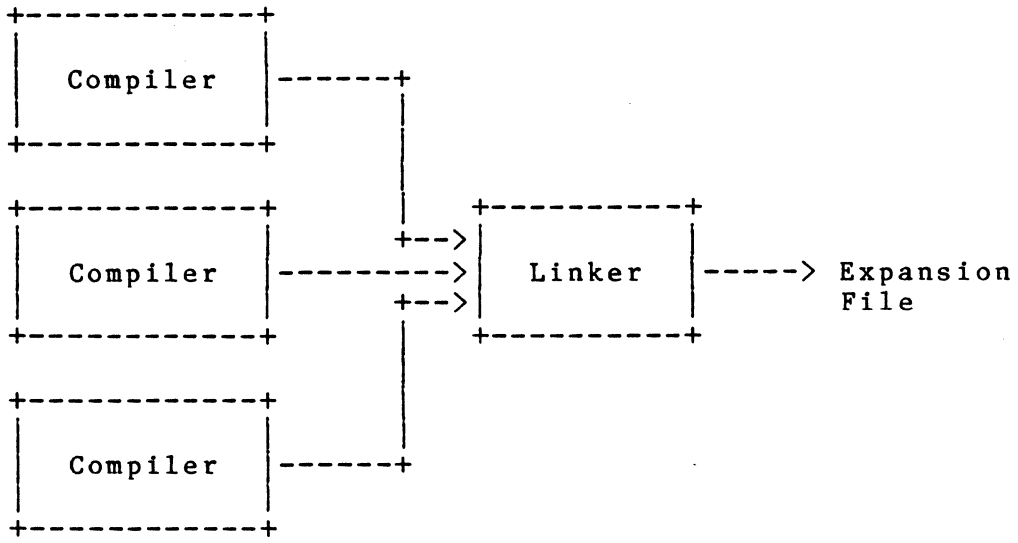
Separate compilation is the process of compiling portions of a design independently and then linking them together to form a complete design. If a change is made to a design, only those drawings involved in the change need to be recompiled. Once compiled, these drawings are then linked with the rest of the compiled drawings to form the complete design. In general, the process of linking takes less time than compiling. This means that separate compilation and linking is faster than compiling the design as a single piece.

The biggest advantage occurs when the design is hierarchical and reuses many drawings. For example, if the design consists of 50 instances of the FOO drawing, the FOO drawing can be compiled once and linked into the design in the 50 places where it is used. This process is probably 25 times faster than compiling the design completely.

Separate compilation in the SCALDsystem is supported by the Compiler and Linker. The Linker produces an expansion file that is identical to the expansion file produced by the Compiler if the design is compiled as one piece. Therefore, there are two paths that can be used to compile designs:



Separate Compilation



5.18 HOW DOES SEPARATE COMPILATION WORK?

The Compiler operates in one of two modes: normal compilation, where the entire design is compiled as a single piece, and separate compilation, where the design can be compiled in portions which are linked together to form the complete design.

When compiling a design as a single piece, the name of the highest level drawing is given (the ROOT drawing). The Compiler compiles the root drawing and all drawings used by the root drawing and so on until all drawings have been compiled. The result is an expansion file that describes the entire design. When a change is made to any drawing in the design, the entire design must be recompiled.

When separately compiling a design, the name of the drawing to be compiled is specified. However, the design itself also must be named so that the Compiler and Linker know to which design the drawing belongs. The concept of a design is central to separate compilation. The design is the name given to a collection of drawings that together form some circuit. The design name corresponds to the ROOT name when compiling the design as a single piece.

When compiling the design FOO as a single piece, the following command is used:

```
compile foo
```

This assumes the existence of the compiler.cmd file that contains compiler directives. The ROOT_DRAWING directive in the compiler.cmd file is ignored and ROOT_DRAWING FOO; is assumed. The Compiler produces an expansion file for the entire FOO design.

When separately compiling the design foo, the following command is used:

```
seplink foo
```

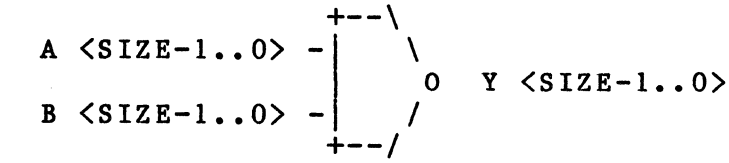
Again, a compiler.cmd file is assumed to exist. This file compiles (separately) all modules (drawings) which are needed by the design "foo," and, if no fatal errors have occurred, links them together. Necessary modules to perform this compilation are automatically determined for each drawing (for example, 'ABC').The ROOT_DRAWING ABC; directive overrides whatever ROOT_DRAWING directive appears in the directives file. The Compiler's output is an expansion file describing the drawing ABC and must be linked with the rest of the design FOO to create a complete expansion file for FOO.

Once all of the pieces of the design have been compiled, they can be linked together with the SCALD Linker. The Linker reads each of the individually compiled portions of the design, determines how they fit together, and creates the expansion and synonym files. The output of the Linker has the same format and contents as the expansion and synonym files produced by the Compiler when the design is compiled as a single piece (it is generally impossible for a program to tell whether the expansion file was produced by the Linker or the Compiler).

The Linker always produces an expansion file and a synonyms file.

5.19 LIMITATIONS ON SEPARATE COMPILATION

Drawings that use parameters are difficult to handle when performing separate compilation. This is because the parameters may have different values everywhere the drawing is used. Take, for example, the SIZE parameter. An LS00 (nand gate) may be given the SIZE parameter to specify the number of bits it represents. The LS00 body is defined as follows:



Notice that the width of each pin depends on the value of SIZE. The TIME model for the LS00 is, therefore, parameterized. This means that the drawing depends upon some parameter (in this case SIZE). If the LS00.TIME drawing is compiled separately, it cannot be linked wherever the LS00 is used because each instance may involve a different value of the SIZE parameter.

The SCALD Compiler detects when a drawing is parameterized and will not permit one to be separately compiled. A drawing that is not parameterized and, therefore, separately compilable, is called a module. The Linker links modules together to form a complete design. It operates under the assumption that a module can be used anywhere in the design without change. This is true because modules are not parameterized.

When the Compiler is compiling a module, it must decide which of the bodies in the module correspond to other modules. These are not compiled since they are to be linked together with the Linker. Bodies that are not modules are compiled out flat. These include all plumbing bodies (such as MERGE or NOT) and all parameterized bodies (such as those that use the SIZE parameter). A module can correspond to one drawing or to many, depending upon how much parameterization occurs and how many plumbing bodies there are.

The result of the compilation is an expansion file for the drawing specified, with all parameterized portions compiled out. Modules may be referenced that are to be linked in by the Linker.

The following is a summary of the restrictions placed on separate compilation.

1. Only non-parameterized drawings may be compiled. In particular, this means that few (if any) library components can be separately compiled since they typically use the SIZE parameter.

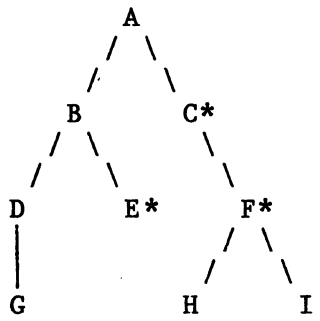
2. The Compiler detects parameters and flags an error if a parameterized drawing is separately compiled.

3. All plumbing and parameterized drawings are compiled completely just as though separate compilation was not being performed.

4. All pages of a drawing must be compiled together. This means that a flat design gains nothing from separate compilation since the entire design is a single drawing.

A SEPARATE COMPILATION EXAMPLE

The following example shows a design hierarchy. The root of the design, the design's name, is 'A'. The other drawings in design 'A' are 'B', 'C', 'D', 'E', 'F', 'G', 'H', and 'I'. The drawings marked with an '*' are parameterized; that is, they use one or more parameters such as SIZE or DELAY. The parameterized drawings are 'C', 'F', and 'E'. For purposes of discussion each drawing is assumed to be represented by the file '<drawing>.LOGIC.1.1'. For example, the drawing 'A' is represented by the file 'A.LOGIC.1.1'.



When the 'A' design is separately compiled, six modules are created: 'A', 'B', 'D', 'G', 'H', and 'I'. For each compiled module, there are four pieces of information kept: the expansion data, the synonyms data, a list of the files that were compiled to produce the module, and the list of modules used by the module that were not compiled. These are shown below:

Module name	Files compiled to produce module	Modules used but not compiled
A	A.LOGIC.1.1 C.LOGIC.1.1 F.LOGIC.1.1	B
B	B.LOGIC.1.1 E.LOGIC.1.1	D
D	D.LOGIC.1.1	G
G	G.LOGIC.1.1	---
H	H.LOGIC.1.1	---
I	I.LOGIC.1.1	---

Note that the drawings 'C', 'E', and 'F' are not modules. Note further that the files for these drawings are included in the files for the 'A', 'B', and 'A' modules respectively. When the Compiler compiles the 'A' drawing, it also compiles the 'C' and 'F' drawings since these are not modules; they are parameterized. If the user tries to compile the 'C', 'F', or 'E' drawings, the Compiler will generate an error message.

5.20 THE CONCEPT OF THE DESIGN

Separate compilation implies the notion of a design. This is the name given to the entire circuit being created and comprises all of the drawings. Currently, the design name and the root drawing name of the design must be the same.

Whenever a drawing is separately compiled, the name of the drawing and the name of the design to which the drawing belongs must be specified. This is done automatically by the SEPLINK command. This action is required to allow the SCALDsystem to organize and keep track of the separately compiled pieces of the design so that they may be linked together.

Currently, a design has no structure; that is, a design may not comprise other designs. The design is managed in the same directory in which the compilations take place. This will be discussed further below. Drawings may be part of more than one design, and more than one design can be compiled and processed in the same directory.

5.21 HOW TO SEPARATELY COMPILE AN EXISTING DESIGN

An existing design that has never been separately compiled can be easily set up for separate compilation. The user simply links the design. The Linker determines which drawings have not been compiled (which at the start is all of the design's drawings) and compiles them automatically.

For example, assume that a design exists with the root drawing IBOX. To set up the design for separate compilation and to get an expansion file for the design, use the command:

```
seplink IBOX
```

The Linker starts by searching for the root of the given design (IBOX) which is IBOX. If IBOX has not been compiled, the Linker calls the Compiler to compile it. A check is made to see if the modules used by the IBOX module are up to date. In this case, they do not even exist, so the Linker

starts the Compiler to compile each module. Once these modules are compiled, they are checked to see if the modules THEY use are up to date. If not, these modules are compiled. This process continues until all modules that make up the design IBOX are compiled. At this point, the modules are linked together to form an expansion file (cmpexp.dat) and a synonyms file (cmpsyn.dat) for the design IBOX.

From this point on, the user simply types the command

```
seplink IBOX
```

to get a new compiled version of the design.

5.22 HOW TO DETERMINE WHAT NEEDS TO BE COMPILED

Once a design is set up for separate compilation (all of the modules in the design have been separately compiled), the system can easily handle changes in the design.

Consider, for example, a design whose name is FOO comprised of the modules FOO, A, B, and C. FOO is the root and it uses the other modules. Assume that a drawing within module A is edited. The user now wishes to create an expansion file for the design FOO. This is done with the command:

```
seplink FOO
```

The Linker first checks to see if each of the modules is up to date. In this case, it finds that the drawing files from which module A is created have been modified. It therefore calls the Compiler to recompile module A with the command:

```
sepcomp FOO A
```

Once this is complete, all modules are up to date and the Linker links them together to produce the output expansion file.

5.23 DRAWING TYPES

The Compiler ALWAYS compiles with a specific destination in mind. The COMPILE directive is used to specify what type of compilation is to be performed. When compiling for the Timing Verifier, the directive COMPILE TIME; is used. When compiling for the Packager, the directive COMPILE LOGIC; is used and so on. Compile types are used in separate compilation as well.

If the FOO design is to be timing verified, simulated, and packaged, it must be separately compiled for TIME, SIM, and LOGIC respectively. To link a design, the design name and compile type should be specified. For example, the FOO design can be linked together to create an expansion file for the Timing Verifier with the command:

```
seplink FOO TIME
```

The Linker will check to make sure that all of the modules in the FOO design have been compiled for TIME, compile those that have not, and then link the design together. If the compile type is not specified, the Linker assumes LOGIC.

5.24 SEPARATE COMPILATION DATA BASE

This section mentions the notion of a "current directory". On UNIX and VMS, this is self-explanatory. The interpretation of this information for CMS follows in the next section.

The compiler output for each separate compilation must be kept around so that it is available for the linker when linking is done. This is done by maintaining a hierarchical file structure within the current directory. This structure is contained within a subdirectory (of the current directory) called "complink". As long as all SEPCOMP and SEPLINK commands for a design are done from the same directory, all of the compiled modules for that design can be found by SEPLINK. Three additional commands (RMCOMP, SHOWCOMP, and GETCOMP) exist to remove compilations from this structure, show what drawings have been compiled, and to get the compiler output for a specified compilation so that it can be examined. For example, suppose you run:

```
seplink foo
```

where the design "foo" contains a drawing "bar". To examine what happened when "bar" was compiled, the command:

```
getcomp foo bar
```

is used. Once this command has been issued, the compiler output for "bar" can be examined from the current directory. This makes it unnecessary to know the details of the separate compilation data base to use separate compilation and get at the results of any of the compilations that have been done.

The data base also maintains the linker files for each design. To examine what happened when "foo" was linked, the command:

```
getlink foo
```

is used. Once this command has been issued, the linker output for "foo" can be examined or used from the current directory. Note that "getlink" gets you the results of the last "seplink" performed on that design for a particular compile type, while "seplink" updates the results (if necessary) and makes them available within the current directory.

The user can see what drawings have been compiled with the "showcomp" command. For example, to see what drawings have been compiled for the "foo" design, the commands:

```
showcomp foo all
showcomp foo all TIME
showcomp foo all SIM
     -or-
showcomp foo all all
```

can be used. All of the drawings for the specified design and compile type are shown.

If the drawing "bar" becomes obsolete and is no longer in design "foo", then the commands:

```
rmcomp foo bar
rmcomp foo bar time
rmcomp foo bar sim
      -or-
rmcomp foo bar ALL
```

remove it from the data base, assuming that these are the compilations that have been done on this drawing. (If "foo" has been linked for LOGIC, TIME, and SIM, it is likely that each of these compilations will exist for a given drawing.)

5.25 SEPARATE COMPILATION DATA BASE ON CMS

Since there are no hierarchical directories on CMS, all files "in" the separate compilation data base are in the current directory. All of the commands available on UNIX and VMS are available on CMS with slight changes in interpretation. Commands that make files available within the current directory (on UNIX) copy files from their data base names to their standard names on CMS. For example, running:

```
getcomp foo bar
```

on CMS causes the compiler output for the LOGIC compilation of drawing "bar" for design "foo" to be copied to the files:

```
CMPEXP DATA A
CMPSYN DATA A
CMPLOG DATA A
CMPLST DATA A
```

if they are found. Those that are not found will not (of course) be copied, and the standard file for that data will not exist.

Since all files are accessible within the current directory, it is recommended that the following command be used in lieu of "getcomp" to look at the results of a compilation. Executing:

```
showcomp -files foo bar
```

causes the showcomp command to list the drawing foo AND the files which contain the results of that compilation. They can then be examined directly. This saves having to copy them.

5.26 DETAILS OF SEPARATE COMPILATION COMMANDS

There are three commands used to compile and link designs. They are compile, sepcomp, and seplink. There are three additional commands associated with separate compilation maintenance at the drawing level (rmcomp, showcomp, and getcomp) and similar commands for maintenance at the design level (rmlink, showlink, and getlink). These will be described below. In the following, letter case is not important for command arguments. That is, the argument may be in upper case, lower case, or mixed case. If the argument contains "special" characters (characters other than letters and digits), it should be placed in quotes. Command names should be in lower case.

The command line arguments in the following commands replace directives in the directives file (compiler.cmd). The compiler.cmd file is always read. If a command argument appears for the ROOT_DRAWING or COMPILE directive, the corresponding directives are ignored in the directives file. An empty argument ("") is considered the same as if the argument was not specified.

compile <drawing name> <compile type>

This command is used to run the Compiler to compile a complete design. The output of the Compiler is used directly without processing by the Linker. The output is not saved in the separate compilation database. <drawing name> is the name of the drawing to be

compiled. It replaces the the ROOT_DRAWING directive. If this argument does not appear or is null, then the ROOT_DRAWING directive specified in the compiler.cmd file is used. <compile type> specifies the type of compilation to be performed. It replaces the COMPILE directive. If this argument is null or omitted then the COMPILE directive (in the compiler.cmd file) is used. LOGIC is assumed if omitted in both places. Some example compile commands:

compile foo
: Compile the drawing "foo". Compile type is specified by the COMPILE directive in the compiler.cmd file. If it does not appear, LOGIC is assumed.

compile "My Drawing" TIME
: Compile the drawing "My Drawing" for TIME. The ROOT_DRAWING and COMPILE directives (if they exist) in the compiler directives file are ignored.

compile "" SIM
: Compile the drawing specified by the ROOT_DRAWING directive in the compiler directives file for SIM. Ignore any COMPILE directive in the directives.

compile "thisdrawing ""
: Compile the drawing "thisdrawing" ignoring any ROOT_DRAWING directive in the compiler directives file. The compile type is specified in the compiler directives file (with the COMPILE directive).

sepcomp <design name> <drawing name> [<compile type>]

This command is used to separately compile a module within a design. The modules so compiled can be linked together by the Linker to produce an expansion file for use elsewhere in the SCALDsystem. <design name> is the name of the design of which the specified drawing is a part. The design name MUST be the same as the name of the root drawing of the design and must ALWAYS be specified. <drawing name> is the name of the drawing (module) to be compiled. This drawing is a part of the specified design. The drawing name must ALWAYS be specified. <compile type> is the type of compilation that is to be performed. If unspecified, LOGIC is assumed. Some example SEPCOMP commands:

sepcomp foo "my Drawing"
Separately compile the drawing "my Drawing" which is a part of the "FOO" design. Compile for LOGIC.

sepcomp designone designone sim
Separately compile the drawing "designone" which is a part of the "designone" design (and is the root as it has the same name). Compile for SIM.

seplink [-f] [-forcecompile] <design name> [<compile type>]

This command is used to run the Linker to link the design creating an expansion and synonyms file. <design name> is the name of the design to be linked and is the same as the design name used when separately compiling. It must be specified. <compile type> is the compilation type. If unspecified, the Linker assumes LOGIC. The Linker calls the Compiler to compile portions of the design that are not up to date. The linker is run if any drawing has been changed and no drawing contained errors affecting the integrity of the separate compilation. The "-f" flag forces linking to occur (following the compilation of any out-of-date drawings) regardless of the the state of the design or any errors that occurred in compilation. The "-forcecompile" flag causes all drawings, whether or not they are up-to-date, to be compiled. Some example link commands are:

seplink foo
Check all drawings used in design foo and compile those found to have been changed since their last compile for LOGIC. If no drawings have severe errors, then make the linker results available in the current directory either by (1) linking the design (if any drawing has changed since the last link) or by (2) making the results from the last link available in the current directory (if no drawings have changed since then).

seplink -f "ABC" TIME
Check all drawings used in design ABC and compile those found to have been changed since their last compile for TIME. Link the design "ABC" for TIME regardless of whether or not it seems necessary.

getcomp [-list] [-log] [-exp] <desn name> <draw name> [<comp type>]

This command is used to retrieve the compiler output for the separate compilation specified. If a SEPCOMP has been done for the specified arguments in this directory, then a GETCOMP will make the results of that compilation available in the current directory. They can then be examined. This is especially useful when many separate compilations have automatically been done for the design in response to a SEPLINK command. <desn name> is the name of the design of which the specified drawing is a part. The design name must ALWAYS be specified. <draw name> is the name of the drawing (module) to be retrieved. This drawing is a part of the specified design. The drawing name must always be specified. <comp type> is the type of compilation to be retrieved. If unspecified, LOGIC is assumed. The "-list" flag requests the retrieval of the compiler listing (CMPLST) file. The "-log" flag requests the retrieval of the compiler log (CMPLOG) file. The "-exp" flag requests the retrieval of the compiler expansion (CMPEXP) and synonyms (CMPSYN) files. If no flags are specified then all of the above files are retrieved. Some example GETCOMP commands:

getcomp foo "my Drawing"
Gets the output files for the last time the "my Drawing" drawing of the "foo" design was compiled for LOGIC.

getcomp -list designone designone sim
Gets the listing file for the last time the "designone" drawing of the "designone" design was compiled for SIM. This allows the compiler errors for drawing designone to be viewed.

getlink [-log] [-exp] <design name> [<compile type>]

This command is used to retrieve the linker output for the design specified. If a SEPLINK has been done followed by other actions on other designs, then a GETLINK will make the results of that linking available again in the current directory. They can then be examined or used. <design name> is the name of the design whose results are to be retrieved. It must ALWAYS be specified. <compile type> is the type of compilation to be retrieved. If unspecified, LOGIC is assumed. The "-log" flag requests the retrieval of the linker log (LNKLOG) file. The "-exp" flag

requests the retrieval of the linker expansion (CMPEXP) and synonyms (CMPSYN) files. If no flags are specified, then all of the above files are retrieved. Some example GETLINK commands:

```
getlink "my Design"
```
Gets the output files for the last time "my Design" was linked for LOGIC.

```
getlink -log designone sim
```
Gets the linker log file for the last time "designone" was linked for SIM. This allows the examination of the linker errors for that design.

showcomp [-files] <design name> <drawing name> [<compile type>]

This command is used to display the contents of the separate compilations data base. <design name> is the name of the design of which the specified drawing is a part. The design name must ALWAYS be specified. <drawing name> is the name of the drawing (module) (from that design) to be shown. The drawing name must ALWAYS be specified. <compile type> is the type of compilation that was performed. If unspecified, LOGIC is assumed. Wildcards are used to determine what is to be displayed. The wildcards are "all" or "*" (both of which mean ALL of that thing). The "-files" flag causes the listing of the files used to capture the compiler/linker results as well as the names of the drawings and designs themselves. Some example SHOWCOMP commands:

```
showcomp foo all
```
Shows the drawings that are part of the design "foo" that have been compiled for LOGIC.

```
showcomp -files "foo" all TIME
```
Shows the drawings that are part of the design "foo" that have been compiled for TIME. Also lists the files holding the Compiler and Linker results for design "foo"

```
showcomp "foo" all all OR
showcomp "foo" "*" "*"
```
Shows the drawings that are part of the design "foo" that have been compiled for any type. They are summarized by compile type.

```
showcomp all all all
```

For each design in the data base, all drawings for all compilation types are displayed. They are summarized by compile type.

showlink [-files] <design name> [<compile type>]

This command is used to display the contents of the separate compilations data base at the design level. <design name> is the name of the design of which the specified drawing is a part. The design name must ALWAYS be specified. <compile type> is the type of compilation that was performed. If unspecified, LOGIC is assumed. Wildcards are used to determine what is to be displayed. The wildcards are "all" or "*" (both of which mean ALL of that thing). The specified designs are displayed, showing the specified compile types that have been done. If the <compile type> is unspecified, LOGIC is assumed. The "-files" flag causes the listing of the files used to capture the linker results as well as the names of the designs themselves. Some example SHOWLINK commands:

showlink foo all
Shows all compile types that exists in the data base for design "FOO".

showlink -files "foo" TIME
Indicates whether or not compile type TIME exists (in the data base) for design "FOO" and lists the linker output files for this design and type if there are any.

showlink all all
Shows all designs and compile types in the data base.

rmcomp [-list] [-log] [-exp] <design name> <drawing name> [<comptype>]

This command removes the specified separate compilation from the separate compilations data base. This separate compilation will no longer exist (in the current directory) and must be redone before the drawing can be linked. <design name> is the name of the design of which the specified drawing is a part. The design name must ALWAYS be specified. <drawing name> is the name of the drawing (module) to be fully or partially removed. This drawing is a part of the specified design. The drawing name must ALWAYS

be specified. <comptype> is the type of compilation for which the removal is to occur. If unspecified, LOGIC is assumed. The "-list" flag requests the removal of the compiler list (CMPLST) file. The "-log" flag requests the removal of the compiler log (CMPLOG) file. The "-exp" flag requests the removal of the compiler expansion (CMPEXP) and synonyms (CMPSYN) files. If no flags are specified, then all files and the drawing itself (for that compile type) are removed. Wildcards ("all" and "*") can be used to remove larger amounts of data. Some example RMCOMP commands:

rmcomp foo "my Drawing"
Remove from the separate compilation data the LOGIC separate compilation done on the drawing "my Drawing" which is a part of the "FOO" design.

rmcomp designone all all
Remove all separate compilations concerning the design "designone" from the separate compilations data base in the current directory. The design "designone" will also be removed.

rmcomp -log all all all
Remove the log file from all drawings in all designs and for all compile types. (This can be done to conserve disk space.)

rmcomp all all all
Remove the entire separate compilations data base (ALL designs) from the current directory.

rmlink [-list] [-log] [-exp] <design name> [<comptype>]

This command is similar to the rmcomp command except that it operates at the level of designs and linker output files rather than at the level of drawings and compiler output files. <design name> is the name of a design. It must ALWAYS be specified. <comptype> is the type of compilation type to be fully or partially removed. If unspecified, LOGIC is assumed. The "-log" flag requests the removal of the linker log (LNKLOG) file. The "-exp" flag requests the removal of the linker expansion (CMPEXP) and synonyms (CMPSYN) files. If no flags are specified, then all files and all drawings and the design itself (for that compile type) are removed. Wildcards ("all" and "*") can be used to remove larger amounts of data. Some example

RMLINK commands:

rmlink foo
: Remove from the separate compilation data all compilations done for LOGIC for the design "FOO".

rmlink designone all
: Remove all separate compilations concerning the design "designone" from the separate compilations data base in the current directory. The design "designone" will also be removed.

rmlink -log foo all
: Remove the log file from the linker results for all compile types of design "FOO". (Note that linker error listings would no longer be available for design "FOO" unless it was linked again.)

rmlink all all
: Remove the entire separate compilations data base (ALL designs) from the current directory.

5.27 ESCAPING WILDCARDS AND NAMES BEGINNING WITH "-"

This section applies to the separate compilation commands only. It does not apply to the "compile" command.

Many of the separate compilation commands take flags and some take wildcards. All of the flags begin with the character "-" and anything beginning with that character is considered to be an attempt to specify a flag. Design and drawing names beginning with "-" or matching one of the wildcards must be treated specially to show that they are not flags or wildcards. This is done by proceeding the name with the flag argument "-". The following examples illustrate this.

```
rmcomp -list -exp time
```
Remove the list, expansion, and synonyms files from the data stored for drawing "time" for compile type LOGIC.

```
rmcomp -list - -exp time
```
Remove the list file from the compilation data stored for drawing "-exp" for compile type TIME.

```
rmcomp -log all all all
```
Remove the log file from the compiler results for all drawings of all designs and for all compile types.

```
rmcomp -log - all all all
```
Remove the log file from the compiler results for all drawings of design "all" and for all compile types.

```
rmcomp -log - - all all
```
Remove the log file from the compiler results for all drawings of design "-" and for all compile types.

Selection Expressions for SCALD Drawings

5.28 INTRODUCTION

Parameters attached to bodies can be used to "customize" the body's drawing. The most common parameter is SIZE which is used to specify the number of bits represented by the body. Another commonly used parameter is DELAY. The use of parameters allows the designer to have each instance of a body represent a slightly different implementation without having many different bodies and drawings.

In some cases, there is a need for radical differences between implementations of a single circuit. These differences are not simply parametric, which can be handled easily with body parameters, but involve changes in the circuit topology. For example, a gate may be designed with several different versions, one version having input protection diodes, one with internal pullups, one with high capacitance load drive capability, and others with combinations of these. Since each version represents the same gate, the user would prefer to define a single body representing the gate and then use some parameter to control which of the gate representations is used. This is supported in the SCALDsystem by drawing versions and selection expressions.

5.29 DRAWING VERSIONS

Each different implementation of a single circuit is called a drawing version. There is no limit to the number of versions that can be defined for a single drawing. Each of the drawing versions corresponds to the same body. In the gate example described above (a NAND gate), the following drawings might be created:

```
NAND.BODY.1.1
NAND.LOGIC.1.1   <- "generic" NAND representation
NAND.LOGIC.2.1   <- NAND with input diodes
NAND.LOGIC.3.1   <- NAND with internal pullups
NAND.LOGIC.4.1   <- NAND with high-C drive
        .
        .
        .
```

Four versions are shown above (though more can be defined) and each of the versions shown has only one page (though a drawing version can have any number of pages).

5.30 SELECTION EXPRESSIONS

If a drawing has more than one version, there must be a method of selecting which one is to be used for any particular instance. This is done with the use of parameters. When the NAND body is placed in a drawing, a parameter must be attached that specifies which of the NAND drawing versions is to be used to allow each instance of the body to refer to a different implementation.

Once the parameters have been attached to the bodies, there must be a method of selecting the appropriate drawing version. This is done with a selection expression. The selection expression defines the "context" in which the drawing is valid. In the case of the NAND, the context is used to select which of the drawing versions (or NAND gate implementations) is to be used.

A selection expression can be an arbitrary integer or Boolean expression. In the NAND example, assume that the parameter TYPE is used to select among the drawing versions. Four selection expressions are needed; one for each drawing version:

```
NAND.LOGIC.1.1     (TYPE=0)
NAND.LOGIC.2.1     (TYPE=1)
NAND.LOGIC.3.1     (TYPE=2)
NAND.LOGIC.4.1     (TYPE=3)
        .
        .
        .
```

The selection expressions (TYPE=0, TYPE=1, ...) define for which values of the parameter TYPE a particular drawing version is to be used. The user sets the value of the TYPE parameter to select the desired version of the NAND.LOGIC drawing.

Selection expressions are defined in the drawing as the EXPR property which must be attached to the DRAWING body of the drawing. If a drawing has more than one version, each version must be given a selection expression to specify under what conditions that version is to be used.

5.31 HOW SELECTION EXPRESSIONS ARE EVALUATED

Whenever a component is found that has more than one implementation (more than one drawing version), the Compiler must decide which version is to be used. This is done by evaluating the selection expressions for each version and picking the version whose selection expression evaluates TRUE.

If the selection expression is a Boolean expression, as in SIZE>2, the selection expression is TRUE if the expression evaluates TRUE. If the selection expression is an integer expression, as in SIZE+1, the selection expression is TRUE if its value is not 0. If the selection expression is empty (or absent), it evaluates to TRUE.

Only one version's selection expression may evaluate TRUE for any given instance. An error is generated if the Compiler finds that the selection expressions for more than one version are TRUE. For example, the following selection expressions are in error because two of them evaluate TRUE for SIZE=2:

```
(SIZE>1)
(SIZE=2)
(SIZE<=1)
```

In the above examples, the selection expressions have been shown using the SIZE parameter although any parameter or text macro may be used in a selection expression.

If the Compiler discovers an error when evaluating selection expressions, it will output the selection expressions for all of the versions to make it easier for the user to see what has happened and to provide a guide to solving the problem.

5.32 SELECTION EXPRESSIONS IN DRAWINGS

As mentioned above, the selection expression is defined by the EXPR property attached the DRAWING body of the drawing. It is important to note that the EXPR property must appear in the first page of the drawing. If the drawing has more than one page, the first page is the lowest numbered page; the first page need not be numbered 1.

The EXPR property in the first page of the drawing defines the context for the entire drawing. Once the drawing has been selected, further selection can be performed. If EXPR properties are used in other pages of the drawing, the Compiler will evaluate them to decide if that page is to be used. This gives the user the ability to define a selection expression for the entire drawing, and to specify a second selection expression for each page to determine whether it is used.

5.33 EXPRESSION EVALUATION

Selection expressions follow the standard SCALDsystem expression evaluation rules. Only integer expressions are supported for selection expressions. There are five built-in functions:

NOT(x)	logical NOT of the operand X
ORD(x)	ordinal value of the operand X
ABS(x)	absolute value of the operand X
MIN(x,y,...)	minimum value of the operands X,Y,...
MAX(x,y,...)	maximum value of the operands X,Y,...

There are two logical operators:

a OR b	logical OR of A, B
a AND b	logical AND of A, B

There are six relational operators:

<	less than
<=	less than or equal
=	equal
<>	not equal
>=	greater than or equal
>	greater than

There are five arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
a MOD b	"remainder" of A / B

The operator precedence is given in the following chart. The highest precedence is given first. Operators with the same precedence are shown on the same line.

```
* / MOD
+ -
< <= = <> >= >
AND
OR
```

The consequences of this ordering are important for evaluating expressions without parentheses to describe the evaluation order. The following examples show expressions in two forms. The first without parentheses. The second has parentheses added to show how the expression would be evaluated. Parentheses can always be added to override default evaluation order. When there are several operators

with the same precedence, evaluation is from left to right.

```
SIZE>2 AND SIZE<4
(SIZE>2) AND (SIZE<4)

3+4*10=5/3+2
(3+(4*10)) = ((5/3)+2)

1+2-3+4
(((1+2)-3)+4)

SIZE+1>3<43+X=0
(((SIZE+1)>3)<(43+X))=0
```

5.34 A NOTE TO 6.0 SCALDsystem USERS

In the 6.0 release of the Compiler, an error is generated if the EXPR property does not appear in each drawing version. This was to prepare the users for the 7.0 release. The 7.0 Compiler will continue to process the 6.0 style selection expressions (using MENU bodies in version 1 of the drawing) only if the drawings are in the old format (not written with the 7.0 Graphics Editor). If the drawings are to be modified, or if complete compatiblity with the 7.0 release is desired, the following steps should be taken:

1. Make sure that the EXPR property appears attached to the DRAWING body in the first page of each of the drawing versions. This property should define the selection expression for that drawing version.

2. Delete version 1 of the drawing. This version contains the MENU body and is no longer needed. MENU bodies are not supported in the 7.0 Compiler for new drawings.

3. Make sure that the first page (containing the EXPR property) for each drawing version has been written with the 7.0 Graphics Editor. This can be done by reading and writing it as part of checking for the existence of the EXPR property. No other pages of the drawing need to be changed or written with the 7.0 Graphics Editor.

5.35 EXPRESSION BNF

The following is the BNF for expressions in the Compiler.

```
<expression> ::= <Boolean expression> |
                 <expression> <Bool OP> <Boolean expression>

<Bool OP> ::= OR | XOR

<Boolean expression> ::=
                  <relational expression> |
                  <Boolean expression> AND <relational expression>

<relational expression> ::=
                  <simple expression> |
                  <simple expression> <rel OP> <simple expression>

<rel OP> ::= < | > | <> | = | >= | <=

<simple expression> ::= <term> |
                        <sign> <term> |
                        <simple expression> <add OP> <term>

<sign> ::= + | -

<add OP> ::= + | -

<term> ::= <factor> |
           <term> <mul OP> <factor>

<mul OP> ::= * | / | MOD

<factor> ::= <unsigned constant> |
             ( <expression> ) |
             NOT <factor> |
             ABS ( <expression> ) |
             ORD ( <expression> ) |
             MIN ( <expression> , <expression> ) |
             MAX ( <expression> , <expression> )

<unsigned constant> ::= <unsigned number>
```

5.36 COMPILER ERROR MESSAGES

One of the purposes of the Compiler is to detect problems in the user's design. There are several problems that are detected and for each a message is produced by the Compiler. The message is intended to inform the user of what problem has been encountered, where it is in the design, and its severity. In addition, the Compiler will print an explanation of each message generated. These messages are extracted from this document. The purpose of this document is to explain the messages produced by the Compiler and, for each, to give some suggestions about how to correct the problem.

5.37 CLASSES OF PROBLEMS

Problems detected by the Compiler are assigned a particular severity. This is done to emphasize important problems (those that cause the design to not function) and de-emphasize those that can wait. There are three classes of problems:

1. Errors

   An error is a problem that must be fixed before further progress can be made. For example, a missing drawing would be considered an error since some portion of the design is missing. Some errors are fatal. That is, their presence causes the Compiler to stop. For example, if the user specifies no drawing to be compiled, the Compiler will generate an error message and stop.

2. Oversights

   An oversight is a problem that is not so severe as to cause the design to be obviously non-functional. For example, if there are no PATH properties on bodies, the Compiler generates oversight messages. The design will work, but, since the path names will be assigned by the Compiler, they will not be easily interpreted by the user. This may make the design difficult to work with. It is also dangerous since such path names may not be consistent between Compiler runs. In general, oversights may be ignored for a while, but should be fixed eventually.

3. Warnings

   A warning is a problem that is very minor. For example, if a default value for a parameter is used, the Compiler generates a warning message. There is no reason to remedy this condition since the Compiler will always produce the same output.

5.38 FORMAT OF MESSAGES

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

```
#n ERROR(m): <message>
#n OVERSIGHT(m): <message>
#n WARNING(m): <message>
```

where <n> indicates how many of the particular class of message have occurred so far, <m> is the message code, and <message> is the text of the message. For example:

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

is the 287th error found in the compile (lots of problems here), the error code is 22 indicating "String length exceeded" which means some string (a quoted sequence of characters) is longer than the maximum allowed (255).

Following the message are several lines describing the drawing in which the error was detected, the body in the drawing, the pin of the body, etc. These are intended to specify the location of the error as accurately as possible to simplify finding and correcting the problem.

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

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

5.39 SPECIAL MESSAGE REPORTING

Some problems are detected while reading input files. These usually are syntax problems (for example, the text was mistyped). For these errors, the Compiler prints the text being read and points to the position in the text where the problem was detected. For example, given the signal name:

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

an error #20 will be displayed as follows:

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

The error really occurs earlier where the '[' was used instead of '{'. It is, in general, impossible for the Compiler to accurately determine the true position of the error; the pointer is always at the position where the error was detected.

If an error occurs in an expanded text macro, the Compiler prints the input line then prints it again with the expanded text macro inserted. For instance, given the signal:

```
FOO <SUBSCRIPT> \G \R 2
```

and the text macro SUBSCRIPT = '0..-12', the following will be printed when the compiler detects the error:

```
'FOO' <SUBSCRIPT> \G \R 2
'FOO' <0..-12> \G \R 2
          ^
#1 ERROR(16):  Bit value invalid
```

Only the text macro being currently read is expanded. Notice also that the signal name (FOO) is in quotes. The Compiler quotes signal names internally to make them easier to read. The quotes are included in the error output so that the designer can make sure that the compiler has interpreted the signal name portion correctly.

With some problems, the pointer into the line points not to the position where the error was detected but to the position where the error occurred. For instance, given the signal:

```
FOO <0..BARF>
```

with BARF undefined (never defined as a text macro or a parameter) the Compiler produces the following:

```
'FOO' <0..1 BARF>
            ^
#1 ERROR(59): identifier has not been declared

'FOO' <0..1 BARF>
           ^
#2 ERROR(11): expected >
```

The Compiler knows it found an undefined text macro when it was expecting either the end of the bit subscript or another element in the bit list. Since it got neither, it points to the beginning of the identifier and indicates that it wanted a '>' to end the subscript. Whenever the error message starts with 'expected', the Compiler prints a pointer to the position where it wants what it expected.

5.40 SUMMARY OF MESSAGES BY NUMBER

The rest of this document contains an ordered (by message code) listing of the Compiler messages and some hints and suggestions about the causes of each problem and how to recover. Whenever the message is "special" (causes the input line being read to be displayed as described above), it is so noted.

Each error number is listed below with one of the following:

1. Error message text.

2. Oversight message text.

3. Warning message text.

4. "Not used". This means that the error message number is available for future use.

5. "Reserved". This means that the error number is used for debugging or other Valid internal operations.

ERROR #1: Expected identifier

This error is generated whenever the Compiler is expecting an identifier (a string of letters, digits, or '_' starting with a letter) and finds some other data. Identifiers are used as names in properties, text macros, and as operands for Compiler directives. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #2: Expected =

This error is generated whenever the Compiler is expecting an equal (=) and finds some other data. Equals are used in many places: between property names and values, in expressions, and in the FILETYPE specification at the beginning of data files. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #3: Unused.

ERROR #4: Unused.

ERROR #5: Expected ,

This error is generated whenever the Compiler is expecting a comma (,) and finds some other data. Commas are used to separate elements in list and are required, for example, in the argument list for the MIN and MAX functions. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #6: Unused.

ERROR #7: Expected)

This error is generated whenever the Compiler is expecting a right parenthesis ( )) and finds some other character. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #8: Unused.

ERROR #9: Wrong file type for text macros

This error is generated whenever the Compiler finds a file with an incorrect FILETYPE specification. All SCALDsystem data files have a FILETYPE at the beginning to allow each program to verify the file contents. In this case, the text macro file being read has the wrong file type Make sure the correct file was specified. The correct file type is FILETYPE=TEXTMACROS. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #10: Expected <

This error is generated whenever the Compiler is expecting a less than character (<) and finds some other character. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #11: Expected >

This error is generated whenever the Compiler is expecting a less than character (<) and finds some other character. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #12: Expected ;

This error is generated whenever the Compiler is expecting a semi-colon (;) and finds some other character. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The Compiler will continue to correctly compile the input even though there was a missing semi-colon. A semi-colon is required as a means of clearly delimiting the end of a command or data item. When some error occurs, the Compiler searches for the semi-colon in order to find the beginning of the next item to read. This helps the Compiler successfully recover from an error.

ERROR #13: Expected :

This error is generated whenever the Compiler is expecting a colon (:) and finds some other character. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #14: Unexpected symbol in integer expression

This error is generated whenever the Compiler is reading an expression (such as in a selection expression or a bit subscript) and finds something unexpected. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. When this error occurs, the compiler is expecting one of the following:

1. A constant.
2. An expression in parentheses, e.g. (2+3).
3. NOT followed by an item from this list.
4. A function (ORD, ABS, MAX, MIN).
5. An identifier whose value is one of the above (such as a text macro whose value is an integer expression such as (23+56) or a parameter whose value is an integer, e.g. SIZE = 1).

ERROR #15: Expected (

This error is generated whenever the Compiler is expecting a left parenthesis (( ) and finds some other character. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #16: Bit value invalid

This error is generated whenever the Compiler is reading a bit subscript and finds an illegal bit value. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The bit value is printed as well. Bit values are invalid if they are negative or are greater than the largest allowed bit number. Since the largest allowed bit number is 231-1 (2147483647) this error usually means that the bit value is negative. This error is most likely to occur

when specifying a bit with an expression. For example, FOO<SIZE-2..0> when SIZE=1.

ERROR #17: Unused.

ERROR #18: Unused.

ERROR #19: Unused.

ERROR #20: Unmatched closing comment character

This error is generated when the Compiler encounters a closing comment character (}) without a matching starting comment character ({). The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Either this symbol is extraneous or the beginning of the comment was never specified. If the symbol really is extraneous, the compiler continues the compilation with no further errors. If it isn't, bogus errors will probably have been generated as the compiler tried to read the text of the comment.

ERROR #21: Unused.

ERROR #22: String length exceeded

This error is generated as the Compiler is reading a string and finds that the string is too long. Strings are limited to 255 characters. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The string is truncated at the current position (pointed to by the compiler) and the compiler reads until it finds the closing quote or the end of the input line. Make the string shorter!

ERROR #23: Illegal character found

This error is generated when the Compiler finds an illegal character in an input file. All non-printing characters except TAB are illegal. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Remove the character.

ERROR #24: Expression value overflow

This error is generated when the Compiler evaluates an expression whose value overflows. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. An overflow does not cause the Compiler to abort; it assigns the value 0 to the result (unless it knows a more reasonable value) and continues with the compilation.

ERROR #25: Division by zero

This error is generated when the Compiler detects division by 0 during evaluation of an expression. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Division by 0 does not cause the Compiler to abort; it skips the division and continues with the compilation.

ERROR #26: Unused.

ERROR #27: Unused.

ERROR #28: Unused.

ERROR #29: Unused.

ERROR #30: Unexpected symbol in bit subscript

This error is generated when the Compiler finds unexpected characters in a bit subscript. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The symbols expected by the Compiler in a bit subscript are:

1. A subrange symbol (..).
2. A colon (:) specifying a bit step.
3. A comma (,) specifying start of next element in a bit list.
4. A greater than symbol (>) indicating the end of the subscript.

ERROR #31: Unknown REPORT specification

This error is generated when the Compiler finds a report specification for the REPORT directive that is unknown by the Compiler. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The most likely cause is a mis-type of the report name.

ERROR #32: Non-printing character found

This error is detected when the Compiler is reading characters from an input file. A non-printing character has been. This is not permitted. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #33: Expected a string

This error is detected when the Compiler is expecting a string (a quoted sequence of printing characters) and finds some other data. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Strings are expected in the following places, among others:

1. In signal names: a signal property must have a property value specified as a string.
2. In compiler directives: the file names for input directories must be specified as strings.
3. In compiler directives: the name of the root drawing for the compilation must be specified as as a string.
4. In the text macro file: the definition of each text macro must be specified as a string.

ERROR #34: Comment not closed before end of input

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

ERROR #35: Specified parameter # > allowed # params

This error is generated when the Compiler is processing text macro parameters and finds a parameter number greater than 4. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Text macros may have up to 4 parameters. Each parameter is referenced by %n where <n> is the parameter number. This error is generated when <n> exceeds the maximum.

ERROR #36: Signal MUST have high assertion char

This error is generated when the Compiler finds a signal that does not have an explicit assertion specification. The signal syntax can be configured so that there are characters to be used to specify whether the signal is high-asserted or low-asserted. If a character is specified for both of these, one must be used for every signal. If no high-assertion character is to be used, the signal syntax configuration should be changed to make the high-assertion character NULL (''). The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The Compiler directive PERMIT_NO_ASSERT ON; can be used to cause the Compiler to allow the absence of an assertion character and to default to high-asserted.

OVERSIGHT #37: Expected .

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

ERROR #38: File name has already been specified

This error is generated whenever the Compiler finds a file name, in a list of files, specified more than once. This can happen with the PROPERTY_FILE, TEXT_MACRO_FILE, and MASTER_LIBRARY directives. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Check the file list and remove duplicate entries.

ERROR #39: Undefined identifier in expression

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

OVERSIGHT #40: Expected END

This warning is generated when the Compiler reaches what it expects to be the end of a file and no END is found. An END must be present at the end of the directives, text macro, and property attributes files (among others). The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The END is used to inform the compiler that the file is complete and that it isn't unfinished or missing some text. This is not a fatal.

OVERSIGHT #41: Identifier length exceeded

This oversight is generated when the Compiler encounters an identifier (a string of letters, digits, or '_' starting with a letter) that has more than 16 characters. Identifiers are used as names in properties and text macros. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The compiler ignores the rest of the identifier and continues with

the compilation.

ERROR #42: Reserved.

ERROR #43: Text macro parameter exceeds max length

This error is generated when the Compiler encounters a text macro parameter whose definition exceeds 16 characters (the maximum permitted). The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Reduce the length of the parameter.

ERROR #44: Constant width value out of range

This error is generated when the Compiler finds an illegal width specification for a signal constant. An illegal width is one that is <= 0 or greater than the maximum allowed signal width (2147483647). The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. When this error occurs, it is probable that the constant specified is simply ridiculous since the compiler is capable of representing a very large constant internally. If the constant being specified is supposed to be huge, it may have to be created in pieces that are concatenated together. For instance, a 1000 bit constant 0 may be generated as follows: 0(250) : 0(250) : 0(250) : 0(250) or (more simply): 0 r 1000.

ERROR #45: Directive has already been specified

This error is generated when the Compiler finds a directive that is used more than once. Most directives can only appear in the directives file once. Some exceptions are PROPERTY_FILE, DIRECTORY, and LIBRARY. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Delete the extra directive.

ERROR #46: Duplicate global text macro definition

This error is generated when the Compiler finds a global text macro definition for a text macro that has already been defined. Global text macros are defined with the text macro file. The text macro may have

been already defined by the Compiler, the standard SCALD text macro file, or one of the user's text macro files. The Compiler prints the input line along with a pointer to the Check to make sure the text macro name is spelled correctly and check the rest of the text macro file for duplications. See the text macro documentation for a complete list of predefined text macros.

ERROR #47: Invalid specification for inheritance

This error is detected when the Compiler is processing a property attributes file and detects an illegal or unknown specification for the INHERIT attribute. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The INHERIT attribute can be specified with PIN, BODY, and/or SIGNAL. See the compiler property documentation for a complete description.

ERROR #48: Unknown property attribute

This error is detected when the Compiler is processing a property attribute file and finds an unknown attribute specification. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Among the supported property attributes are: INHERIT, PARAMETER, FILTER, and PERMIT. See the property documentation for a complete description of the use of the property file.

ERROR #49: Unused.

ERROR #50: Unused.

ERROR #51: Unknown compiler directive

This error is generated when the Compiler is reading the directives file and encounters an unknown Compiler directive. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. For a complete list of the compiler directives, see the compiler directive documentation. This error will not (normally) prevent the compiler from reading the rest of the directives.

ERROR #52: Invalid specification for directive

This error is detected when the Compiler is processing a directive from the directives file and it encounters an invalid operand. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. There are several directives that require an operand (such as WARNINGS which takes either ON or OFF as its operand) rather than a string (such as ROOTDRAWING). If the operand for any of these directives is not what the Compiler expected, this error is generated. See the Compiler directives documentation for a complete description of the directives.

ERROR #53: Input line exceeds maximum length

This error is detected when the Compiler tries to read a line greater than its 255 character input buffer. The input line (up to the point the Compiler;s buffer was filled) is printed with a pointer to the last character in the buffer (just in case the line is filled with ' '). The input line will need to be broken before the Compiler can read it. The compiler's input buffer is 255 characters but an 80 character maximum input line is advisable (so that compiler output can be easily viewed on a TTY).

ERROR #54: Unused.

ERROR #55: Wrong file type for property attributes

This error is generated when the Compiler finds that the specified property attributes file has the wrong file type. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Each SCALDsystem data file is identified by its FILETYPE specification which appears as the first line in the file. The FILETYPE for property attributes is ATTRIBUTES.

ERROR #56: Text macro parameter cannot be found

This error is generated when the compiler is expanding a text macro which refers to a parameter (as in %1) which cannot be found. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The name of the text

macro and the missing parameter number are both printed. Make sure that all text macros that expect parameters are given them and that each parameter is delimited with a space.

ERROR #57: End of input before end of expression

This error is generated when the Compiler finds the end of input before the end of the expression being evaluated. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. If the line printed is blank (empty) the input line was null. This occurs, for instance, when an empty string is given as a parameter to a text macro that expects and integer parameter (such as r or w). It can also occur when a null value is given to a property with the INTEGER(PARAMETER) attribute.

ERROR #58: Extraneous characters at end of expr

This error is generated when the Compiler finds extra characters at the end of an expression. All characters in a string (such as a property value) that are to be evaluated as an expression must be part of the expression. When this error is generated, the Compiler has found some characters that do not form part of a legal expression. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #59: Identifier has not been declared

This error is generated when the Compiler finds a reference to an indentifier (a string of letters, digits, or '_' starting with a letter) that has not been defined. Identifiers are used as names in properties, text macros, and as operands for Compiler directives. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. If the identifier is supposed to be a defined text macro, check the DEFINE bodies to make sure it was correctly defined. If it was supposed to be a parameter of the body, check the body definition to make sure that it was correctly defined there. This error is fatal to separate compilation.

ERROR #60: Unused.

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

This error is generated when the Compiler is processing a constant signal name that has an explicit radix specification that is not permitted. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Signal constants may be specified in any base from 2 to 16. All other bases are illegal. The default base is 2.

ERROR #62: Extraneous junk at end of menu version

This error is generated when the Compiler is processing the VERSION property of a MENU body and finds its value (which should be a constant) to contain unexpected characters. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Check the menu definition. One of the version specifiers is incorrect. Correct it.

ERROR #63: Extraneous junk at end of Boolean expression.

This error is generated when the Compiler is processing the EXPR property of a MENU body and finds its value (which should be an expression) to contain unexpected characters. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Check the menu definition. One of the expression specifiers is incorrect. Correct it. Check the menu body and correct the selection expression. Refer to the drawing and body definition documentation for a description of the form for a selection expression.

ERROR #64: Max text macro recursion depth exceeded

This error is generated when the Compiler finds that a text macro refers to itself and recurses too deeply. The Compiler prints the current text macro definition with a pointer to the position where the error was detected. The drawing name and the name of the text macro being expanded are also printed. Recursive text macros are dangerous and should be avoided. Correct the definition of the given text macro.

ERROR #65: Compile extension name is too long

This error is detected when the Compiler is processing the COMPILE directive and finds a specification that is too long. A limit of 11 characters is placed on a drawing type. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Change the type given in the directive and recompile. For a more complete description of the COMPILE directive, see the Compiler directives documentation.

ERROR #66: Compilation to .PRIM files not permitted

This error is detected when the Compiler is processing the COMPILE directive and finds that PRIM is specified. PRIM is not permitted as a COMPILE type. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The PRIM extension is used to specify a primitive part regardless of whether the compilation is to produce output for timing verification, simulation, or the Packager. The PRIM extension cannot be used here. Some correct extensions are: LOGIC, SIM, and TIME. See the Compiler directives documentation for a complete description of this directive. See the SCALD overview documentation for a description of drawing extensions.

ERROR #67: Reserved.

WARNING #68: Library file has already been specified

This warning is generated when the Compiler is processing the LIBRARY directive and finds a library specified that was previously specified. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #69: Reserved.

ERROR #70: Version number is outside allowed range

This error is generated when the Compiler finds an illegal value for a drawing versions. Drawing versions are allowed to be from 1 to 10000. The name of the drawing is printed. The version number for this drawing should be changed.

ERROR #71: Page number is outside allowed range

This error is generated when the Compiler finds an illegal value for a drawing page. A drawing may not have more than 10000 pages. The name of the drawing is printed.

ERROR #72: Duplicate page number

This error is generated when the Compiler finds that a page of a drawing is duplicated (appears more than once in the SCALD directories). This occurs by accidentally writing a page of the drawing to more than one SCALD directory. One of the copies should be deleted. The other way in which this error occurs is by having two drawings with the same name (even though they are different drawings). The name of one of the drawings will have to be changed.

ERROR #73: Reserved.

ERROR #74: Reserved.

ERROR #75: Reserved.

ERROR #76: Boolean expression already defined for this version.

This error is generated when the Compiler is processing a MENU body and finds a selection expression specified for a drawing version that already had an expression assigned for it. Check the VERSION and EXPR properties of all of the MENU bodies to make sure that each drawing version is referenced only once.

ERROR #77: Specified version is not in directory

This error is generated when the Compiler is processing a MENU body and finds a VERSION property referring to a version of the drawing that does not exist. Check the VERSION property to make sure that it was correctly entered. Check the SCALD directory to make sure that the desired drawing version exists.

ERROR #78: Only MENU bodies are allowed here

This error is generated when the Compiler is reading version 1 of a drawing with more than 1 version and finds a body that is not a MENU body. Version 1 of a multi-version drawing must contain the MENU body and no other.

ERROR #79: Expected a version 1 with MENU body(s)

This error is generated when the Compiler finds a drawing with multiple versions but without a version 1. Version 1 must exist and contain a MENU body when the drawing has multiple versions.

OVERSIGHT #80: Illegal property on MENU body

This oversight is generated when the Compiler finds a property other than EXPR or VERSION on a MENU body. The Compiler associates the value of property EXPRj with the version specified by the property value of VERSIONj. Any other property name here is an oversight. Check for a misspelling.

ERROR #81: Illegal MENU property number

This error is generated when the Compiler is processing EXPRj and VERSIONj properties on MENU bodies and finds j to be outside the range 1 to 256. The compiler associates the value of property EXPRj with the version specified by the property value of VERSIONj. Check the property names for a mistype.

ERROR #82: Same MENU expression property found twice

This error is generated when the Compiler finds an EXPRj property specified more than once on a MENU body. The compiler associates the value of property EXPRj with the version specified by the property value of VERSIONj. Check the MENU body definition to make sure that the property names are correct.

ERROR #83: Expected signal name or constant

This error is generated when the Compiler is reading a signal and finds it to be malformed. A signal must have a name string or be a signal constant. If the Compiler finds neither of these, it generates this error. The Compiler prints the input line along with

a pointer to the position in the line where the problem was detected. The most likely cause of this error is the illegal use of reserved characters are quoted). Check the signal name and correct it.

ERROR #84: Replication factor is out of range

This error is generated when the Compiler is processing a signal and finds a replication property value that is illegal. The value of this property is outside the allowed range (that is, it is <= 0 or greater than the maximum allowed number of bits in a signal). The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The maximum number of bits in a signal is 2147483647 so this error usually means that the replication specified is <= 0.

ERROR #85: Expected FILE_TYPE specification

This error is generated when the Compiler starts reading an input file and does not find a FILETYPE specification. All SCALDsystem data files are identified with a FILETYPE specification at the beginning to allow programs to check their inputs. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The Compiler skips the rest of the offending file. The directives file is the only Compiler input file that does not need a FILETYPE specification. Check the given file to make sure that it is the correct file. If it is, add the proper FILETYPE specification.

ERROR #86: File is not of the correct type

This error is generated when the Compiler finds that an input file has the wrong FILETYPE specification. All SCALDsystem data files are identified with a FILETYPE specifier which allows the programs to check the validity of input data. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The Compiler skips the rest of the offending file. The directives file is the only Compiler input file that does not need a FILETYPE specification. Check the given file to make sure that it is the correct file. If it is, change its FILETYPE specification.

ERROR #87: Directory file name previously specified

This error is generated when the Compiler finds a SCALD directory has been specified more than once with a DIRECTORY directive in the directives file. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Directory files should be specified only once.

ERROR #88: Unused.

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

This error is generated when the Compiler finds that a string (a quoted sequence of characters) does not have a closing quote before the end of the current line is reached. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Check the line to make sure that the string is correctly specified. All strings must be closed. This most commonly occurs when a quote is accidentally placed into a signal name.

ERROR #90: Reserved.

ERROR #91: Unused.

OVERSIGHT #92: Invalid (warnings and oversights only)

This oversight is generated when the Compiler finds an error message code number specified as an operand of the SUPPRESS directive. Only oversight and warning messages may be suppressed. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected.

ERROR #93: Expected directory file name

This error is generated when the Compiler is processing a DIRECTORY directory and does not find a directory file name where it expected one. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. This error may occur for three reasons:

1. The file name may simply have been forgotten.
2. The file name may not have been put in quotes.
3. A comma was found after a file name indicating another file name was to follow in a list of files but another file name was not found.

ERROR #94: Invalid value for print width

This error is generated when the Compiler is processing the PRINT_WIDTH directive and finds an illegal print width value specified. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Print width values must lie in the range 80 to 132.

OVERSIGHT #95: Drawing has not been written w/ 5.5 GED

This oversight is generated when the Compiler finds drawing that has not been written with the 5.5 or later Graphics Editor. The proper processing of inheritable pin properties requires the Compiler to understand the graphical representation of nets within the drawing. This is becauses pin properties inherit along graphical structures and not along electrical ones. For example, the signal X may have an inheritable pin property in one part of the drawing but may not have it on another part of the same drawing. They are electrically connected because they have the same name (X) but are not connected for purposes of inheriting pin properties properties. Versions 5.5 and later of the Graphics Editor pass Graphical information to the Compiler so that inheritable pin processing can be performed correctly. If a drawing is read by the Compiler which has NOT been written by the 5.5 or later GED, it generates this oversight and continues. It WILL process the drawing correctly but will use MORE memory and take longer. All drawings should be written with the new Graphics Editor.

ERROR #96: Bit subscript on constant not permitted

This error is generated when the Compiler is processing a constant signal and finds a bit subscript. Constants may not be given bit subscripts.

The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. There are two mechanisms for specifying the widths of constants:

1. A replication factor may be used such as: 0r 32
2. An explicit width specification may be used such as: 0(32).

OVERSIGHT #97: Unknown output file name

This oversight is generated when the Compiler finds an illegal specification for the OUTPUT directive. The valid OUTPUT directive operands are: LIST, CHIPS, EXPAND, and SYNONYM.

ERROR #98: Unused.

ERROR #99: Reserved.

ERROR #100: Reserved.

ERROR #101: Drawing path name is missing closing)

This error is generated when the Compiler is reading the PRIMITIVE directive and finds a path name without a closing ')'. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. A path name is delimited with '(' and ')'; both must appear.

ERROR #102: Reserved.

ERROR #103: Library not found in master directory

This error is generated when the Compiler finds a library specified with the LIBRARY directive that does not appear in the master library. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. All libraries that are to be accessible with the LIBRARY directive must be listed in the master library file.

ERROR #109: Body properties are not allowed here

This error is generated when the Compiler finds an illegal property on a body. The property name is printed along with the body name. The given property does not have the body permission attribute. That is, it is not allowed to be attached to a body. Check the drawing to make sure that the property is attached to the correct object. Check the property attributes file if the property is supposed to have the body permission attribute.

ERROR #110: Undefined text macro (null value)

This error is detected whenever a text macro is used that has a null definition. The text macro name is printed. An undefined text macro definition may result from some input error. The text macro expands as a null and the compile continues. Check the other error messages and the DEFINE body to correct the definition. This error is fatal to separate compilation.

ERROR #111: No directory was specified

This error is detected at the end of reading the compiler directives file. All of the directives have been read in but no DIRECTORY directive was found informing the Compiler of the directories to use in the compilation. Since the Compiler needs at least one directory, such a condition is an error. Change the compiler directives file to specify the directories to be used. The directories are the same ones used by the graphics editor.

ERROR #112: Separate AND single drawing compilation

This error is generated when the Compiler finds that the SEPARATE_COMPILE and SINGLE_DRAWING directives have both been set to ON. This is not permitted. These two compilation modes are incompatible. One of the directives must be eliminated. Check the directives file and delete one of the directives. The SINGLE_DRAWING directive is only useful when preparing a hierarchical design for processing by an interface program for transfer to some system OTHER THAN THE SCALDsystem.

ERROR #113: Replication is not permitted on pin name

This error is generated when the Compiler finds a pin name with a r replication used within the body drawing. Pin names are not allowed to have signal replication. Remove the replication specification.

ERROR #114: Text macro has already been defined

This error is detected when the compiler is reading a DEFINE body. Two text macros of the same name have been found. The text macro name is printed along with the drawing in which the DEFINE body appears. All text macro names (within the same drawing) must have unique names. Check the DEFINE bodies for duplicate names.

ERROR #115: Same MENU version property found twice

This error is detected when the compiler is reading a MENU body. It found the same version property (VERSIONn) specified more than once on the MENU body. The drawing name and the property name are both printed. Check the MENU body definition to make sure that the property names are correct.

ERROR #116: Expanded string exceeds max string len

This error is generated when a text macro is expanded that exceeds the maximum internal string length. This is most likely to happen when recursion or iteration is used. The number of characters needed to represent the entire string is too long to be represented. The text macro name is printed along with its definition. Correct the definition. This expansion only occurs when property values are being processed for output.

ERROR #117: Textmacro and parameters exceeds max len

This error occurs when the compiler is trying to expand a text macro and its parameters. The resulting string is too large for the internal string representation. This error can occur wherever a text macro is used. Check the definition of the text macro to be sure that any recursion or iteration is being used correctly.

ERROR #118: Expression value is empty

This error is detected when the compiler is processing the value for an integer expression. The value given is NULL (empty). This error occurs when processing integer parameters on bodies or the value specified for the signal properties r and w. The name of the parameter or property whose value is empty is also printed. Check the property or parameter name and correct it. Make sure that parameters to all text macros (such as r and w) are delimited with spaces (that is, a space must appear at the beginning and end of each text macro parameter).

ERROR #119: NC is not permitted as a pin name

This error is generated when the Compile finds a body with a pin name called NC. The NC signal is not permitted as a pin name. The name of the drawing where the body was found and the name of the body with the illegal pin name are printed. Edit the body and change the name of the pin.

ERROR #120: Path name exceeds maximum length

This error is generated when the Compiler finds that the path name has exceeded the maximum allowed (255 characters). This happens when the hierarchy gets too deep. Deep hierarchies can be created with recursion. The path name is made up of individual path elements from each drawing in the hierarchy. This is a FATAL error. That is, the appearance of this error will prevent the compilation from continuing. To correct, reduce the depth of the drawing hierarchy or shorten the path elements (e.g., reduce the length of the drawing abbreviations). The most common cause of excessively long path names is the use of recursive drawings.

ERROR #121: Path element name exceeds maximum length

This error is detected when the compiler creates a path element for a given drawings. If this error occurs, there is a serious problem. Either the drawing abbreviation or the path property is excessively long. FIX IT! The drawing name, the using drawing name, and the drawing's PATH property are printed. If this error occurs it is quite likely that error #120 will also appear. The maximum length

of a path element is 255 characters.

ERROR #122: Drawing has incompatible extensions

This error is detected as the compiler is reading a directory. A drawing with two extensions that are incompatible has been found. The drawing name as well as the two extensions for the drawing are printed. Incompatible drawing extensions are, for example, LOGIC and PART. One of these drawings must be removed from the directory. See the documentation on directories for a more complete description of the restrictions and use of SCALD directories.

OVERSIGHT #123: Selection expr and MENU expr mismatch

This oversight is detected when a drawing is read. The compiler checks the selection expression specified in the drawing menu (if the drawing has more than one version) with the selection expression specified in the drawing (as the EXPR property on the drawing's DRAWING body). If the two expressions are not the same, this error is generated. The drawing name, the selection expression used for the directory (the expression defined in the MENU), and the selection expression found in the drawing are printed. Correct the drawing and menu so that they agree. The compiler always uses the menu expression when the two expressions are different.

ERROR #124: Versioned drawings not written with 7.0 GED

This error is generated when the Compiler is processing a drawing with multiple versions. If there is no MENU body, and the first drawing version was written with 7.0 or later GED, ALL drawing versions must be written with the 7.0 or later GED. This is because selection expressions are handled differently (automatically) in release 7.0 and later. Edit and write all of the versions of the drawing with a 7.0 or later version of GED. Make sure that EVERY drawing version has an EXPR property on its DRAWING body so that the Compiler will know what the selection expression should be. This takes the place of the MENU body.

ERROR #125: Pin properties are not permitted here

This error is generated when the Compiler finds an illegal property on a pin. The property name is printed along with the pin name. The given property does not have the pin permission attribute. That is, it is not allowed to be attached to a pin. Check the drawing to make sure that the property is attached to the correct object. Check the property attributes file if the property is supposed to have the pin permission attribute.

ERROR #126: Text macro is not an identifier

This error is generated when the Compiler is expanding text macros in property values. These text macros are identified by a '%' preceding them. All text macro names MUST be identifiers which are strings of letters, digits, and '_' starting with a letter and no more than 16 characters long.

OVERSIGHT #127: ABBREV property not found for the drawing

This oversight is generated when the drawing is found that does not have an ABBREV property. Every drawing must have a macro abbreviation (attached as the ABBREV property to the DRAWING body of the drawing). A NULL drawing abbreviation is allowed. The Compiler creates an abbreviation and continues. The drawing name and the created abbreviation are printed.

ERROR #128: ABBREV value must be letters, digits & _

This error is generated when the compiler finds an ABBREV property value consisting of characters other than letters, digits, or '_'. The compiler creates a new abbreviation for the drawing. The drawing name, the ABBREV property value, and the new abbreviation are printed.

ERROR #129: Menu entry for version is not permitted

This error is detected when the compiler is reading the a MENU. A multi-version drawing has been found. When the Compiler read the menu (in version 1 of the drawing) it found a menu entry for version 1 of the drawing. This is not permitted; version 1 (the menu) is used only for menus and cannot itself appear in the menu. The drawing name is printed. Change the menu to remove the reference to version 1.

ERROR #130: Scalar reference to vector signal

This error is generated when the Compiler encounters a scalar signal (a signal with no bit subscript and therefore 1 bit wide) after previously encountering the same signal as a vector (a signal with a bit subscript and therefore 1 or more bits wide). A signal cannot be both a scalar and a vector. The signal name is printed. Make them both scalars or vectors. The compiler forces the scalar to be a single bit vector.

ERROR #131: Vector reference to scalar signal

This error is generated when the Compiler encounters a vector signal (a signal with a bit subscript and therefore 1 or more bits wide) after previously encountering the same signal as a scalar (a signal with no bit subscript and therefore 1 bit wide). A signal cannot be both a scalar and a vector. The signal name is printed. Make them both scalars or vectors. The Compiler forces the vector to be a replicated scalar.

ERROR #132: Concatenated signal as pin name

This error is generated when the Compiler finds a pin name on a body that is the concatenation of two or more names such as, F<1>:F<2>. Concatenation is not permitted in pin names. The pin name can be fixed by either using a bit list to combine the signals (the above pin name would be F<1,2> or F<1..2>) or using separate pins for each piece of the concatenated signal. The Compiler gives the pin the first name in the concatenated signal and ignores the rest.

ERROR #133: This property has already been defined

This error is generated when the Compiler finds a property attached more than once to the drawing. This error is detected when processing the DRAWING bodies for a drawing. Since a drawing may have more than one drawing body (one on each page, for example) the potential exists for accidentally attaching a property to more than one. Since such properties are cumulative (that is, they don't replace each other but add up), the Compiler informs the user that the property has appeared more than once.

OVERSIGHT #134: Terminal drawing is not a primitive part

This oversight is generated when the Compiler finds a drawing that has no bodies in it. This is called a TERMINAL drawing for the reason that the design expansion terminates here. The Compiler does not allow a drawing to be a terminal unless explicitly told to permit it. There are two ways of giving such permission. The drawing can be made into a primitive part (such as a .PART drawing) or the TERMINAL=TRUE property may be attached to the drawing's DRAWING body.

ERROR #135: Cannot open compiler directives file

This error is generated when the Compiler tries to open the compiler directives files and finds it cannot. Either the file does not exist or it was incorrectly specified. This is a fatal error; the Compiler needs this file. Correctly specify it and recompile.

ERROR #136: Signals cannot be attached to this body

This error is generated when the Compiler finds signals attached some "special" body (such as a DEFINE or DRAWING body) that cannot have signals attached to it. The most probable cause is accidentally creating a user drawing with one of these names. If this is the case, change the name of the drawing. The name of the body is printed.

ERROR #137: Text macro nesting depth exceeded

Text macros are capable of referring to other text macros (and may refer to themselves). Each time a text macro refers to another, the current definition is saved on a stack and the new definition is started. If the stack becomes filled, this error is generated. The most likely cause is the use of a recursive text macro that has no way of terminating (that is, the text refers to itself which refers to itself, ad infinitum). Care must be taken when using recursive text macros to guarantee they will eventually terminate. The maximum nesting depth is 10.

ERROR #138: Cannot open error documentation file

This error is generated when the Compiler cannot open the file containing error documentation. This is usually due to the file being incorrectly bound to the logical file ERRDOC. It may also be due to an access problem (such as the user not having read access to the file).

ERROR #139: More than 1 selection expression is true

This error is detected when the Compiler attempts to read a specific drawing. The drawing of interest has more than one version and, after evaluating all of them, the Compiler discovers that more than one selection expression is true. The name of the drawing being processing and all the selection expressions are printed. Only one selection expression may evaluate true for any given drawing instance. Correct the selection expressions to ensure they are unique.

OVERSIGHT #140: This signal cannot be DECLAREd

This oversight is detected as the Compiler processes DECLARE body within a drawing. Constant signals (that is, signals that are constants) cannot be DECLAREd. The current drawing name as well as the name of the body (DECLARE) is printed along with the signal name. Remove the constant signal from the signal list of the body. See the Compiler signal documentation for a complete description of signal scopes if there is some confusion.

WARNING #141: SIZE property on non SIZE-wide body

This warning is generated if the compiler encounters an instance of a body that is not size dependent, but has a SIZE parameter attached to it. The warning is issued as it is quite likely that the SIZE parameter is superfluous and will be ignored.

ERROR #142: MENU bodies are not supported

This error is generated when the Compiler finds a MENU body in a drawing written with the 7.0 or later GED. This is not supported. Selection expressions are supported in release 7.0 by placing the EXPR property on the DRAWING body of each drawing version. The Compiler reads this property to determine the selection expression for the drawing. If a drawing

with a MENU body (version 1) exists, it should be deleted. Check to make sure that EXPR properties exist and read and write every drawing version with the 7.0 or later GED. This has been designed to make selection expressions easier to use since a summary MENU drawing, which was difficult to use in any case, does not need to be made.

ERROR #143: Illegal rotation on this body

This error is generated when a body with an illegal rotation is detected. Logic symbols have definite rotational semantics. Throughout The SCALDsystem, MSB to LSB ordering (independent of signal syntax bit ordering) is from top to bottom or left to right. For example, a 2 MERGE merges 2 signals into one with the top signal becoming the MSBs. Rotations are restricted to the following: 0 degrees (right), 90 degrees (up), mirror about Y of 0 degrees (left), and mirror about X of 90 degrees (down). All other (180 and 270 degree rotations) are disallowed.

OVERSIGHT #144: This body should not be given SIZE prop

This oversight is generated when the Compiler finds a SIZE property on a body that cannot be given one. This is true if the body has a HAS_FIXED_SIZE or a NEEDS_NO_SIZE property. The name of the body and drawing are printed. Remove the SIZE property from the body.

ERROR #145: Pin name does not exist

This error is generated when the Compiler finds a reference to a pin name (interface signal) that does not exist on the drawing's corresponding body. The drawing in which the error was found (as well as the body to which the signal is connected) are printed along with the name of the signal. Check the body definition to make sure that the pin names have been correctly assigned and check the signal name for misspellings. A list of all of the pin names on the body is printed to make it easier to identify the problem.

ERROR #146: Pin name does not have this bit

This error is generated when the Compiler finds an

interface signal that references a bit that does not exist on the pin (as defined by the body). The name of the drawing in which the error was found (as well as the name of the body to which the signal is connected) are printed along with the name of the signal. Check the body definition and signal name.

ERROR #147: Root drawing is a primitive.

This error is generated when SINGLEDRAWING is ON and the compiler finds that the root drawing has been specified as a primitive. Forcing any drawing to a primitive is unnecessary with this option, and forcing the root to a primitive is not functional under any option. Check the directives (compiler.cmd) file for an ISPRIMITIVE directive on the root drawing name and check for .prim or .part drawings for the root.

ERROR #148: No root drawing was specified

This error is generated when the Compiler finds that no drawing name has been specified for compilation. The Compiler cannot continue without the name of the drawing it is to compile. This error is fatal: it causes the Compiler to stop immediately. Make sure the drawing name has been specified with the ROOT_DRAWING directive in the Compiler's directives file.

ERROR #149: Synonyms must use single assertion

This error is generated when the Compiler finds signals whose assertions differ being synonymed together. This is not permitted. This error is also generated if all of the assertions of signals within concatenated signals being synonymed do not have the same assertions. Synonyms of signals with differing assertions can only be synonymed with the NOT body. This should be used.

ERROR #150: PERMIT attribute value invalid

This error is generated when the Compiler is processing the PERMIT attribute in the property attributes file and finds an unknown specification. The allowed permission attributes are PIN, BODY, and SIGNAL. Check the property attributes file for misspellings.

ERROR #151: This property not permitted on a SIGNAL

This error is generated when the Compiler finds a property (without the PERMIT(SIGNAL) attribute) attached to a signal. It is most likely that the property was accidentally attached to the signal. The name of the drawing, signal, and property are output. Check the attachment of the property within the drawing to make sire that it has been correctly attached. If this property is supposed to be attached to a signal, correct the property attributes file to give the property the PERMIT(SIGNAL) attribute.

ERROR #152: This property not permitted on a BODY

This error is generated when the Compiler finds a property)without the PERMIT(BODY) attribute) attached to a body. It is most likely that the property was accidentally attached to the body. The name of the drawing, body, and property are output. Check the attachment of the property within the drawing to make sire that it has been correctly attached. If this property is supposed to be attached to a body, correct the property attributes file to give the property the PERMIT(BODY) attribute.

ERROR #153: This property not permitted on a PIN

This error is generated when the Compiler finds a property (without the PERMIT(PIN) attribute) attached to a pin. It is most likely that the property was accidentally attached to the pin. The name of the drawing, body, pin, and property are output. Check the attachment of the property within the drawing to make sire that it has been correctly attached. If this property is supposed to be attached to a pin, correct the property attributes file to give the property the PERMIT(PIN) attribute.

ERROR #154: Unused

ERROR #155: Attempt to synonym 0 and 1

This error is generated when the compiler finds the constant signal '0' and the constant signal '1' being synonymed together. The drawing name, the body in the drawing where the signals are connected, and the signals are printed.

Only like constant signals may be synonymed.

ERROR #156: Signal's width cannot be determined

This error is generated when the Compiler finds a pin name, an NC signal, or an UNNAMED signal whose width (number of bits) cannot be determined from context. The name of the signal, the drawing it appears in, and the name of the body and pin to which it is connected are printed. The width of a signal is determined whenever it is connected to a pin whose width is known or is synonymed to a signal whose width is known. In some circuits, notably those involving complex merger structures, it may not be possible for the Compiler to determine the widths of all of the signals. To correct this problem, the user should use SLASH bodies or give the signals names with the proper number of bits.

ERROR #157: Reserved.

ERROR #158: Unused.

ERROR #159: Synonym of unequal width signals

This error is generated when the Compiler finds two signals synonymed that have different widths. The name of the signal, the name of the drawing in which the signal appears, and the name of the body and pin to which the signal is attached are printed. Check to make sure that the signals were not mistyped. The most frequent cause of this problem is the misuse of MERGE bodies. Synonyms are created whenever two signals are connected to the same pin. Check to make sure that, if a wire has more than one signal name, the signals have the same width.

ERROR #160: Cannot SIZE replicate plumbing drawings

This error is generated when the Compiler finds that a plumbing drawing (such as a MERGE or a NOT) is being SIZE replicated in the Compiler (X_STEP <> SIZE). This is not permitted. The name of the drawing is printed.

ERROR #161: 2 signals with timing assertions synonymed

This error is generated when two signals are synonymed that have a Timing Verifier assertion. Such a signal is FOO !C 0-4. This is forbidden since the one of the assertions will be lost following the synonym (since only one of the signals can be the case signal name). The name of the drawing, the body and pin to which the signals are connected, and the signal names are printed.

ERROR #162: Interface and local signals conflict

This error is generated when the Compiler finds a local and an interface signal with the same name. For example, A\L and A\I. This is not permitted. A signal can only have one scope. That is, it must be either interface, local, or global. The name of the drawing, the name of the body and pin to which the signal is attached, and the name of the signal are printed.

ERROR #163: Local and global signals conflict

This error is generated when the Compiler finds a local and a global signal with the same name. For example, A\L and A\G. This is not permitted. A signal can only have one scope. That is, it must be either interface, local, or global. The name of the drawing, the name of the body and pin to which the signal is attached, and the name of the signal are printed.

ERROR #164: Global and interface signals conflict

This error is generated when the Compiler finds an interface and a global signal with the same name. For example, A\I and A\G. This is not permitted. A signal can only have one scope. That is, it must be either interface, local, or global. The name of the drawing, the name of the body and pin to which the signal is attached, and the name of the signal are printed.

ERROR #165: This signal cannot have scope property

This error is generated when the SCOPE property is found on a signal that should not be given one. For example, the presence of the SCOPE property (specified with \I, for example) on a pin name in a body drawing

would cause this error to be generated. The name of the drawing, the name of the body and pin to which the signal is connected, and the signal are printed.

ERROR #166: Unused.

ERROR #167: Cannot open synonyms file for input

This error is generated if the Compiler is unable to open the temporary scratch synonyms file written during signal processing. This temp file contains synonym information which is to be reformatted and output to CMPSYN. This error indicates that something fundamental has broken. Contact a Valid Service Engineer to get help.

ERROR #168: Cannot close file

This error is generated whenever the Compiler is unable to close a file that was open for either read or write. This is usually due to some protection or disc space problem. The name of the file (either the file name itself or a logical file name) is printed. Check the protections for the file and the user to make sure they are compatible.

ERROR #169: Cannot open file for output

This error is generated when the Compiler fails to open a file for output (write). This is usually due to some protection or disc space problem. The name of the file (either the file name itself or a logical file name) is printed. Check the protections for the file and the user to make sure they are compatible.

ERROR #170: Cannot open master library file

This error is generated when the Compiler fails to open the file containing the master list of libraries for the SCALDsystem. This list is used by the LIBRARY directive. The file name is printed. This is usually due to some protection or disc space problem. The name of the file (either the file name itself or a logical file name) is printed. Check the protections for the file and the user to make sure they are compatible.

ERROR #171: Bit subscript increment of 0 not allowed

This error is generated when the Compiler finds a bit subscript increment of 0 in the bit subscript for a signal. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Increments can be any value but 0.

ERROR #172: Bit subscript should be right to left

This error is generated when the Compiler finds a bit subscript whose bits are out of order. That is, they are opposite that allowed. For example, if the standard bit ordering is right to left (n..0) then a subscript of the form 0..5 is an error. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. If a "bit reversed" bit subscript is desired, an increment of -1 can be used. For example, the subscript 0..5:-1 is perfectly legal. The -1 serves to clearly specify that a reversal of the normal ordering is desired.

ERROR #173: Bit subscript should be left to right

This error is generated when the Compiler finds a bit subscript whose bits are out of order. That is, they are opposite that allowed. For example, if the standard bit ordering is left to right (0..n) then a subscript of the form 5..0 is an error. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. If a "bit reversed" bit subscript is desired, an increment of -1 can be used. For example, the subscript 5..0:-1 is perfectly legal. The -1 serves to clearly specify that a reversal of the normal ordering is desired.

ERROR #174: Unused.

ERROR #175: Unused.

ERROR #176: Unused.

ERROR #177: Selection expr for drawing is FALSE

This error is generated if a drawing is found in which every page has a selection expression that evaluates FALSE. This can happen whether there is more than one version of the drawing or not. A selection expression attached to the first page (whichever that is) is used as the selection expression for the entire drawing. However, each additional page of the drawing may have its own selection expression that must evaluate TRUE in order that that page be valid. If all pages have selection expressions that evaluate FALSE, and there is not more than one version of the drawing, this error occurs. Check the EXPR properties on the DRAWING bodies of each drawing page to make sure that they are correct. If they are, check to make sure that the parameters referred to in the selection expression (such as SIZE) are defined properly.

ERROR #178: Max error value must be >= 1

This error is generated when the Compiler finds a zero or negative value specified for the MAX_ERRORS directive. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Make sure the number is positive.

ERROR #179: Extraneous junk at end of signal

This error is generated when the Compiler finds more characters at the end of a signal name: characters that the Compiler cannot recognize as belonging to the signal name. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. Possible causes are:

1. A concatenation character is missing so that the next signal is misinterpreted.
2. A property was misplaced (before the bit subscript) making the compiler think it had found the end of the signal.
3. The junk at the end of the signal really is JUNK!

ERROR #180: Parameter was declared twice

This error is generated when the Compiler finds a property with the PARAMETER attached to a body more than once. This is an error since the Graphics Editor

does not permit this. The name of the drawing, the name of the body to which the property is attached, and the property are printed.

ERROR #181: Cannot access specified drawing directory

This error is generated when the Compiler cannot open a drawing directory specified by a SCALD directory as containing the files for a drawing. Check to see that the specified directory exists and is accessible (readable and executable) by you.

WARNING #182: Drawing title does not match directory

This warning is generated when the Compiler finds that the TITLE property on the DRAWING body does not match the drawing's name. The Compiler prints the input line along with a pointer to the The drawing name (specified as the TITLE property on the DRAWING body) does not match the name given in the directory. The directory name and the name given in the drawing are both printed.

ERROR #183: X_FIRST must be >= 0 (set to 0)

This error is generated when an X_FIRST text macro is found which has a negative value. This is not permitted. The value must be positive or zero. The name of the drawing in which the definition appears is printed. Check and correct the definition. The value is set to 0.

WARNING #184: PIN_EQUIVALENT no longer supported

This warning is generated when the PIN_EQUIVALENT attribute specification is found in the property attributes file. This attribute is obsolete and has been replaced by the INHERIT(PIN) attribute which has the same meaning. The Compiler assumes INHERIT(PIN) instead of PIN_EQUIVALENT and continues. All property attribute files should be updated to use INHERIT(PIN) in place of PIN_EQUIVALENT.

ERROR #185: SIZE must be >= 0 (set to 1)

This error is generated when a SIZE parameter whose value is negative is found. All SIZE parameters must

have positive values (or 0). The name of the drawing and the body to which the SIZE parameter is attached are printed. Correct the SIZE value. The most common cause of this error is an incorrect expression for the SIZE parameter that evaluates to a negative number in some contexts.

ERROR #186: X_STEP must be > 0 (set to 1)

This error is generated when an X_STEP text macro is found which has a zero or negative value. This is not permitted. The value must be positive. The name of the drawing in which the definition appears is printed. Check and correct the definition. The value is set to 0.

ERROR #187: Assertion chk failure: save CMPLOG file

This error is generated whenever the compiler discovers some internal data problem. The compiler is constantly checking to make sure that its internal data is consistent. If it detects some problem, this message is generated. Contact Valid Logic Systems for a work around and/or corrections. This message indicates an internal compiler error and usually cannot be fixed by the user. Save the data that caused the error as it will be very helpful in finding the problem. It is very important that the CMPLOG file be saved (at a minimum). Valid may also request any of the input or output files for the Compiler. Try to be ready to reproduce the problem for the Service Engineer.

ERROR #188: Parameters not permitted on this body

This error is generated when an instance parameter (a body property with \PARAMETER at the end of the property value) is found on a body that cannot be given a parameter. There are several "special" bodies to which parameters cannot be attached. They are: DRAWING, INTERFACE, DECLARE, DEFINE, DIRECTIVE, STRUCTURE, and MENU. Since parameters may only be added when the body is edited, the most common cause of this error is the creation of a user drawing with one of these reserved names.

ERROR #189: Timing assertion not allowed on pin name

This error is generated when a Timing Verifier assertion is found as part of a pin name. A example Timing Verifier assertion is !C 0-4. These may NOT be assigned to pin names or interface signals. The name of the drawing, the name of the body and pin to which the signal is attached, and the signal or pin name are printed.

ERROR #190: No selection expression evaluates true

This error is generated if the Compiler finds that no selection expression evaluates to true (or 1) for a drawing. Since none is true, the compiler has no drawing to compile. The name of the drawing is printed along with the all the selection expressions (if the drawing has more than one version). Correct the selection expressions so that one will be selected. A NULL expression always evaluates true.

ERROR #191: Drawing not found in the directories

This error is generated when the Compiler finds a body that does not have a corresponding logic (or time, sim, ...) drawing. The drawing name is printed. Some possible causes are:

1. The directory containing the drawing was not specified in the compiler directives.
2. The drawing has not been implemented yet.

This error is fatal to separate compilation.

OVERSIGHT #192: PATH name element is not unique

This oversight is generated when the Compiler detects that the path element for a body is not unique within the drawing. The path element is constructed from the page number of the drawing, the PATH property attached to the body, the abbreviation for the body (from the ABBREV property), and the SIZE replication index (X). If the path element is not unique, the path name for the body or signal will not be unique causing severe problems. The compiler makes the element unique by appending an instance identifier (it enumerates the bodies that appear in the drawing). The instance identifier is separated from the rest of the path element by a ':'. This condition cannot occur if the automatic path generation feature of the graphics editor is used and drawing abbreviations have been

specified. To correct, change the path properties of the drawing instances (that were manually assigned) to make them unique. The name of the drawing, body, and the path property are printed.

ERROR #193: No usable extension found for drawing.

This error is generated when the Compiler finds a body that was listed in an appropriate SCALD directory, but does not have a usable extension for the type of compilation being performed. The drawing name is printed. Some possible causes are:

1. The drawing directory containing the drawing was deleted or moved.
2. The drawing directory containing the drawing could not be opened (such as because of a protection violation).
3. The drawing has not been implemented yet.
4. There are no versions of the drawing with legal extensions for the specified compile type. If compiling for LOGIC, legal extensions are LOGIC and PART. Otherwise, if compiling for XXX, legal extensions are LOGIC, XXX, and PRIM.

WARNING #194: Text macro refers to itself (recursive)

This warning is generated when the Compiler detects a macro recursion.

ERROR #195: Cannot open specified attributes file

This error is generated when the Compiler is unable to open a property attributes file. The name of the file is printed. Make sure the file exists. This error can also be due to some protection or disc space problem. The name of the file (either the file name itself or a logical file name) is printed. Check the protections for the file and the user to make sure they are compatible.

WARNING #196: Default value used for SIZE (1)

This warning is generated when a definition for SIZE

is absent and the SIZE parameter is used. The value 1 is assumed. The name of the drawing is printed. SIZE is defined by attaching it to the body.

OVERSIGHT #197: PATH property not found for body

This oversight is generated when the Compiler cannot find a PATH property attached to a body. The PATH property is used to identify a body within a drawing and in the construction of path names which are used to uniquely identify every part and signal in the design. The Compiler automatically generates a PATH property. PATH properties should be assigned so that they may be referenced in the drawings.

ERROR #198: Bit subscript on undefined width pin

This error is generated when the Compiler finds a vectored interface signal referring to a pin with the NWC property. The NWC property means that the width of the pin is not known and must be derived from its context. When referring to such a pin, the ENTIRE signal must be referenced. Since the width is not known, it is not possible to refer to a specific bit. The entire signal can be referenced as though the pin was a scalar. For example, the pin A\NWC is referred to by the interface signal A\I. The name of the drawing, the name of the body and pin to which the signal is attached, and the name of the offending interface signal are printed.

ERROR #199: Pin name conflicts with previous pin

This error is generated when conflicting pin names are found on a body. Pin may have the same (identical) names or may reference different bits of the same pin name. For example, the pin names A<0>, A<1..2>, and A<3> are legal. The pin names A<0>, A<0..2>, and A<3> are illegal since bit 0 of the pin is referenced by two pins. The name of the body and its pins are printed.

ERROR #200: Pin name and signal widths do not match

This error is generated when a signal is connected to a pin whose width is not the same as the width of the signal. Signal widths must match the widths of pins. The name of the drawing, the name of the body and pin

to which the signal is attached, and the name of the signal are printed. Check these signals to make sure their bit subscripts have been correctly defined.

ERROR #201: Signal fails bubble check on this pin

This error is generated when the Compiler finds a signal whose assertion (high or low) does not match the assertion of the pin to which it is attached. A high-asserted signal must be connected to a pin without a bubble. A lowasserted signal must be connected to a pin with a bubble. The name of the drawing, the name of the body and pin to which the signal is connected, and the name of the signal are printed. These checks can be turned off with the BUBBLE_CHECK OFF; directive.

ERROR #202: Pin name with NWC cannot have subscript

This error is generated if a pin name (on a body) is found with both a subscript and the NWC property. The presence of one make the presence of the other meaningless. A subscript defines the width of the pin while NWC specifies that the width is unknown. Remove one of them from the pin name. The name of the body as well as the pin are printed.

WARNING #203: Reserved

ERROR #204: Pin name cannot use signal negation

This error is generated when a pin name that uses signal negation is found. For example, -A or -A. Signal negation is not permitted in pin names. The name of the body and the pin name are printed. Don't do this.

ERROR #205: Cannot open DRAWING file

This error is generated when the Compiler fails to open the file that contains the specified drawing. The name of the drawing and the file name are printed. Make sure the file exists. This error can also be due to some protection or disc space problem. The name of the file (either the file name itself or a logical file name) is printed. Check the protections for the file and the user to make sure they are compatible.

This error does NOT mean that the drawing does not exist in the SCALDdirectories; it was found, but the drawing file cannot be opened.

ERROR #206: Cannot open specified directory file

This error is generated when the Compiler fails to open a SCALDdirectory file. The file was specified with the DIRECTORY directive. The name of the file is printed. Make sure the file exists. This error can also be due to some protection or disc space problem. The name of the file (either the file name itself or a logical file name) is printed. Check the protections for the file and the user to make sure they are compatible.

ERROR #207: Cannot open syntax configuration file

This error is generated when the Compiler fails to open the system-wide signal syntax configuration file. The name of the file is printed. Make sure the file exists. This error can also be due to some protection or disc space problem. The name of the file (either the file name itself or a logical file name) is printed. Check the protections for the file and the user to make sure they are compatible.

ERROR #208: Too many errors in this compile!

This error is generated when the compiler finds too many errors. The error limit is set with the MAX_ERRORS compiler directive. It is set to 1000 by default. After this error is displayed, the compiler halts.

ERROR #209: Cannot open specified text macro file

This error is generated when the Compiler fails to open a text macro file. The file was specified with the TEXT_MACRO_FILE directive. The name of the file is printed. Make sure the file exists. This error can also be due to some protection or disc space problem. The name of the file (either the file name itself or a logical file name) is printed. Check the protections for the file and the user to make sure they are compatible.

ERROR #210: - Primitive cannot have NWC pin

This error is generated when a primitive is found that has a pin with the NWC property. This is not permitted. Primitives are "physical" components in that they have specific characteristics and definitions. The SCALDsystem does not permit such a "physical" component to have a pin whose width is unknown (how are pin numbers assigned to such a pin?). The name of the primitive and the drawing in which it is used are printed. Change the definition of the primitive to remove the NWC pin.

ERROR #211: Unused.

ERROR #212: A pin name cannot be a constant

This error is generated when the Compiler finds a pin name that is a constant. For example, the pin name "2". Such pin names are illegal. The name of the body and pin are printed. Change the name of the pin. For example, the addition of a single prefix character will form a legal pin name.

OVERSIGHT #213: Versioned drawing must have EXPR prop

This oversight is generated when the Compiler finds a drawing that has more than one version and one of the versions does not have an EXPR property attached to its DRAWING body to specify the selection expression. An EXPR property is used to inform the Compiler for which contexts the drawing is valid. If the user is using MENU bodies for the drawings, the selection expression is also specified as a property there. The values of these two EXPR properties MUST be identical. This may seem to be redundant but is necessary to eliminate MENU bodies. If no MENU version was used, then either this version will be the one that is picked (since omitted selection expressions evaluate TRUE) or multiple versions will evaluate TRUE, resulting in an ERROR message.

ERROR #214: String not closed before end of signal

This error is generated when a string is found in a signal name that does not have a closing quote. Strings can appear around the name of the signal, or embedded within it, and are part of signal property

specification (the property value). Check the signal to make sure there are no unmatched quotes.

ERROR #215: Pin name is scalar but used as vector

This error is generated when the compiler finds a vectored interface signal that refers to a scalar pin name. For example, A<0>\I referring to the pin A. The name of the drawing, the body name where the signal was found, and the signal are printed. Fix the interface signal or the pin name; they must be consistent.

OVERSIGHT #216: PART not allowed; COMPILE LOGIC assumed

This oversight is generated when the COMPILE PART directive is found in the directives file. This is not permitted. The Compiler prints the input line along with a pointer to the position in the line where the problem was detected. The COMPILE directive specifies the "type" of primitive to be output by the Compiler. COMPILE TIME causes the Compiler to produce an expansion file in containing only Timing Verifier primitives. COMPILE SIM causes the Compiler to produce an expansion file containing only Logic Simulator primitives. The Compiler considers a primitive to any component with the .PART or .PRIM extension. For this reason, COMPILE PART makes no sense since there is no such thing as "part" primitives. "Logic" primitives, on the other hand, are physical components and have a very obvious meaning. The Compiler assumes COMPILE LOGIC and continues.

ERROR #217: Fatal error(s) encountered - run stopped

This error is generated whenever a fatal error has been encountered that is causing compilation to stop. This is the last message printed by the Compiler. It is used to inform the user that fatal errors have occurred.

ERROR #218: Bit lists are not permitted in pin names

This error is generated when a pin name is found that uses bit lists (list of bits or subranges) in its bit subscript. Pin names may only be given single bit or simple subrange bit subscripts. For example, A<0> or

A<5..0> but not A<5, 4, 3, 2, 1, 0>. The name of the body and offending pin are printed along with the name of the drawing in which the body is used. Correct the pin name.

ERROR #219: DECLARE bodies are no longer supported

This error is generated whenever a DECLARE body is found in a drawing. These are no longer supported. Please contact your Valid Logic Systems Service Engineer for details and assistance.

OVERSIGHT #220: PART_NAME property should not be used

This oversight is generated when the Compiler finds a PART_NAME property being used that does not match the body's name. The PART_NAME is used to change the name of the primitive output by the Compiler from the name of the body (the default) to the name specified by the PART_NAME property value. This feature was added to make it possible to perform strange and wonderful name transformations. It is, in general, a bad idea. The rest of the system needs to know the name of a given body within the drawing so that this information may be conveyed to the Graphics Editor. The best example of this need is for back annotation. If the name of the body has been transformed with the PART_NAME property, it is very difficult to tell the Graphics Editor what body to use. The Packager gets around this problem, but this may not be possible for other programs (such as ones written by the user). This oversight is only produced when the chips file is generated. In this way, it becomes clear during library creation that something is amiss while users of the library are not effected.

ERROR #221: Bodies with NWC cannot expand to parts

This error is generated when a body is found that has a pin with the NWC property and the body expands to primitives. This is not permitted. Bodies with pins with the NWC property are considered to be plumbing bodies. That is, they are used to synonym signals together or to otherwise combine or separate signals. MERGErs, NOTs, and TAPs are all plumbing bodies. These bodies are handled in a special manner that is not compatible with bodies whose definitions include primitives. The name of the body is printed along with the name of the drawing in which the body is

used. Remove the NWC property from all of the body's pin names.

ERROR #222: PRIMITIVE specifies unfound drawing

This error is generated when the Compiler cannot find the drawing specified with the PRIMITIVE directive. The most likely cause is the absence of the proper SCALDdirectory (specified with the DIRECTORY directive). The name of the drawing is printed. Check the directives file to verify that all of the required SCALDdirectories have been specified.

ERROR #223: Unused.

ERROR #224: Unused.

ERROR #225: Unused.

ERROR #226: Unused.

ERROR #227: Unused.

ERROR #228: Unused.

ERROR #229: Unused.

ERROR #230: Unused.

ERROR #231: Unused.

ERROR #232: Unused.

ERROR #233: Unused.

ERROR #234: Unused.

ERROR #235: Signal synonymed to its own complement

This error is generated when the a signal is synonymed its own opposite. For example, (if standard syntax is used) signal A and -A are complements of each other, both of which are high asserted. Signal -A* is equivalent (in polarity) to A, and A* is equivalent (in polarity) to A-, except that -A* and A* are low asserted. Within the SCALD language, A and -A* are the SAME signal and -A and A* are both taken to represent the complement of it. It is illegal to synonym A (or -A*) to -A (or A*) in any way. The error is fixed by finding where the synonyming took place, and correcting this. Often, this is caused by using A* where -A* is actually correct (where a low asserted version of A is desired).

ERROR #236: Unused.

ERROR #237: Unused.

ERROR #238: Unused.

ERROR #239: Unused.

ERROR #240: .PRIM and .PART both found for drawing

This error is generated when both a .PRIM extension and a .PART extension have been defined for the same drawing. This is confusing as both mean the same thing so the compiler does not know which to use. Fix by removing one of these extensions from the drawing. NOTE: earlier versions of the compiler treated these differently, so be sure to keep the one that specifies all necessary drawing properties.

ERROR #241: Signal synonymed to its own complement

This error is generated when a pin is synonymed to its own opposite. For example, a bubbled pin Q* (or Q) is connected to the same signal as the unbubbled version of Q (or Q*) on the same body. (Note that the BUBBLE command of GED can sometimes be used to switch the bubble from an A* pin to an A pin -- the point is that either way, the signals are meant to be complements and therefore should not be synonymed together.) The error message prints the name of the (unbubbled) pin and, if a vector pin, the bits that were synonymed to the same bits of the bubbled version of the pin. The error typically occurs by accidental connection of the wire attached to one pin to the wire attached to the other. It can also occur by inadvertently synonyming one of the attached signals to the other. It is fixed by finding where the two wires have been attached or synonymed, and undoing the connection.