
EMTPWorks

Report Script Language Reference

Chris Dewhurst and Jean Mahseredjian

EMTP®-EMTPWorks

www.emtp.com

Disclaimer

Information in this document is subject to change without notice.

No part of this manual may be copied or reproduced in any form or by any means without written permission from PGSTech.

PGSTech makes no representation or warranty with respect to the adequacy or accuracy of this documentation or the software which it describes. In no event will PGSTech or its direct or indirect suppliers be liable for any damages whatsoever including, but not limited to, direct, indirect, incidental, or consequential damages of any character including, without limitation, loss of business profits, data, business information, or any and all other commercial damages or losses, or for any damages in excess of the list price for the license to the software and documentation.

Copyrights

- EMTP® and EMTPWorks: © Copyright 2016-2021 Hydro-Québec, EDF and RTE.
- DesignWorks: © Copyright 1985-2011 Capilano Computing Systems Ltd., 2011-2021 by Flying Objects Software Inc.
- ScopeView: © Copyright 2010-2021 Hydro-Québec

Table of Contents

Part I – Script Language Reference.....	9
I.1. Command Language Introduction	9
I.1.1 Basic Script Structure	9
I.1.2 EMTPWorks Object Types.....	11
I.1.3 Definition Commands.....	12
I.1.4 Current Design or Current Object	14
I.1.5 Data Types.....	14
I.1.6 Blocks	15
I.1.7 Command Arguments.....	15
I.1.8 Control and Escape Characters	16
I.1.9 Script Variables.....	17
I.1.10 Attribute Field References.....	18
I.1.11 Precedence of Field References in Pin Listings	18
I.1.12 Attribute vs. Variable References	19
I.2. Controlling Report Page Layout.....	20
I.2.1 Setting Page Height and Width	20
I.2.2 Defining a Page Header.....	20
I.2.3 Setting Column Alignment.....	21
I.2.4 Defining a Value Break	21
I.3. Date and Time References.....	22
I.3.1 Raw Date and Time Format.....	22
I.3.2 Date and Time Formatting Commands	22
I.4. Sorting and Merging.....	23
I.4.1 The \$SORT Command	23
I.4.2 Enabling Merging	25
I.5. Implementing Mark as OK in Error Checking Scripts.....	28
I.5.1 How the Mark as OK Function Works	28
I.5.2 Error Bit Functions	28
I.5.3 How the Mark as OK Value is Stored.....	29
I.6. Reporting Power and Ground Nets	30
I.6.1 Specifying Signal Sources	30
I.6.2 Creating Design-Specific Signal Source Fields.....	30
I.7. Script Hierarchy Issues.....	32
I.7.1 Types of Hierarchical Netlists.....	32
I.7.2 The \$HIERARCHY Command	32
I.7.3 Restricting Reporting of Internal Circuits.....	33
I.7.4 Listing Format for Internal Circuits	33
I.7.5 Depth Ordering in Pure Netlists	34
I.7.6 Instance vs. Definition vs. Hierarchical Names	35
I.7.7 Power and Ground Connections in Hierarchy	35

I.8.	File Input and Output	37
I.8.1	File Names and Paths.....	37
I.8.2	Types of Output	38
I.8.3	Text File Input	39
I.9.	Regular Expressions.....	39
I.9.1	The \$REGEXP Command	40
I.9.2	Regular Expression Syntax.....	40
I.9.3	Match Variables	41
I.9.4	Back-references Within an Expression	41
I.9.5	Match Variable References Outside an Expression.....	42
I.9.6	Differences from Unix Regular Expressions.....	42
I.9.7	Regular Expression Examples	42
I.10.	Script Examples	44
I.10.1	EMTP Netlist.rfm	44
II.	Part II – Script Keyword Reference	51
1	\$ALERT1/\$ALERT2	53
2	\$ALIGNCOLSON/\$ALIGNCOLSOFF	54
3	\$AND	54
4	\$ASSIGNINSTNAMES.....	54
5	\$ASSIGNNAMES.....	55
6	\$AUTONUMBER	56
7	\$BLANKREPLACE.....	56
8	\$BREAK	57
9	\$BUSCLOSE	59
10	\$BUSNAME	59
11	\$BUSNAMEON/\$BUSNAMEOFF.....	60
12	\$BUSPINCLOSE	60
13	\$BUSPINNAME	61
14	\$CALLTOOL.....	61
15	\$CHANGECOUNT	62
16	\$CHARMAP	62
17	\$CHECK	63
18	\$CHECKSUM.....	64
19	\$CHILDEMTPPHASE	64
20	\$CHILDSIGNAME.....	65
21	\$CIRCUITNAME	65
22	\$CLEARERRORBIT	66
23	\$CLEARERRORS.....	66
24	\$CLOSECIRCUIT/\$CLOSEDESIGN.....	67
25	\$CLOSEREPORT	67
26	\$CLOSETRANSSCRIPT	67

27	\$COL	68
28	\$COMBDEVSON/\$COMBDEVSOFF	69
29	\$COMBPINSON/\$COMBPINSOFF	69
30	\$COMBSIGSON/\$COMBSIGSOFF	70
31	\$CONTAINS	70
32	\$CONTEND	70
33	\$CONTSTART	71
34	\$COUNT	71
35	\$COUNTINST	72
36	\$COUNTVALUES	72
37	\$CREATEFOLDER/\$CREATEDIRECTORY	73
38	\$CREATEREPORT	74
39	\$CREATETRANSSCRIPT	75
40	\$DATE	75
41	\$DATECREATED/\$DATEDMODIFIED	76
42	\$DEFINEATTR	76
43	\$DEFINEBLOCK	78
44	\$DEFINECIRCUIT	79
45	\$DESIGNNAME	79
46	\$DESIGNPATH	80
47	\$DESIGNPINSIGSOURCE/\$DESIGNSIGSOURCE	81
48	\$DEVCOUNT	82
49	\$DEVICES	84
50	\$DEVINSTNAME	86
51	\$DEVLOC	86
52	\$DEVNAME	86
53	\$DEVPINFORMAT	87
54	\$DEVPINSEQUENCE	88
55	\$DEVSEQ	89
56	\$DEVTOKEN	89
57	\$DIRECTORY	89
58	\$DIV	90
59	\$DWVERSION	90
60	\$ELSE	90
61	\$EMTPPHASE	91
62	\$END	91
63	\$EQ	91
64	\$ERRORBITOFF	91
65	\$ERRORBITON	92
66	\$EVAL	92

67	\$FILEEXISTS.....	93
68	\$FILENAME.....	93
69	\$FIND.....	94
70	\$FOLDER	96
71	\$FULLPATH	96
72	\$GE.....	97
73	\$GRID	97
74	\$GT	97
75	\$HEADER	98
76	\$HEX.....	98
77	\$HIERARCHY	98
78	\$HIERNAMESeparator	99
79	\$IF	100
80	\$INCLUDE	101
81	\$INCLUDEPORTSON/\$INCLUDEPORTSOFF	103
82	\$INLINE	104
83	\$INTERNAL	105
84	\$ISPORT	105
85	\$ISUNCONNPIN	106
86	\$ITEMSEPARATOR	106
87	\$LE	106
88	\$LINESUSED.....	107
89	\$LINEWIDTH	107
90	\$LOWERCASE	108
91	\$LT.....	108
92	\$MAP	109
93	\$MAXITEMSPERLINE	110
94	\$MERGE.....	111
95	\$MINUS.....	112
96	\$MULT	112
97	\$NE	112
98	\$NEWLINE.....	113
99	\$NEWPAGE.....	113
100	\$NONBLANK.....	114
101	\$NOT	114
102	\$NOTES.....	115
103	\$NULL.....	115
104	\$NUMINPS.....	115
105	\$NUMOUTS	116
106	\$NUMPINS.....	116

107	\$NUMEMTPPINS.....	116
108	\$ONEPINSON/ONEPINSOFF.....	117
109	\$OR.....	117
110	\$PAGE.....	117
111	\$PAGELENGTH	118
112	\$PARENTPIN.....	118
113	\$PINDIR	118
114	\$PINNAME.....	119
115	\$PINNUM	120
116	\$PINS.....	120
117	\$PINSEQ.....	121
118	\$PINSIGSOURCE.....	122
119	\$PINTYPE.....	123
120	\$PINTYPEFORMAT.....	123
121	\$PLUS.....	124
122	\$PORTNAME.....	125
123	\$PRIMNAME.....	125
124	\$PROGPATH	126
125	\$PROGRESS	126
126	\$REGEXP	128
127	\$REPLICATE	128
128	\$REPORTON/\$REPORTOFF	129
129	\$REPORTPAGE	129
130	\$SAMEPINCOUNT	130
131	\$SCRIPTPATH	130
132	\$SELECT	131
133	\$SELECTED	131
134	\$SETATTR.....	131
135	\$SETERRORBIT.....	132
136	\$SETSIGWIDTH	133
137	\$SETVAR.....	133
138	\$SIGCOUNT	134
139	\$SIGHIERNAME	135
140	\$SIGINSTNAME	135
141	\$SIGLOC	136
142	\$SIGNALS	136
143	\$SIGNAME	138
144	\$SIGPINFORMAT	139
145	\$SIGSEQ	139
146	\$SIGSOURCE.....	140

147	\$SIGTOKEN	141
148	\$SINGLE.....	142
149	\$SORT	142
150	\$SPACE.....	144
151	\$SYSPIN	145
152	\$SYSTEMOPEN	145
153	\$TAB.....	146
154	\$TABFIELDSON/\$TABFIELDSOFF	146
155	\$TABLE	147
156	\$TEMPPATH	148
157	\$TEXTLINE.....	149
158	\$TIME	149
159	\$TIMECREATED/\$TIMEMODIFIED.....	151
160	\$TYPENAME	151
161	\$UNCONNPINSOFF/\$UNCONNPINSON	152
162	\$UNNAMEDDEVS	152
163	\$UNNAMEDSIGS	152
164	\$UNSELECTEDPINS.....	153
165	\$UNUSEDUNITS	154
166	\$UPPERCASE.....	154
167	\$VERIFY	155
168	\$WRITETRANSSCRIPT.....	155

Part I – Script Language Reference

This manual provides an overview of the Export (Report) script language and then provides reference information on specific Export tool features and applications. Part II – Script Keyword Reference provides detailed information on individual commands. If you are new to writing EMTWorks scripts, you may wish to start by looking at the examples in “Script Examples” on page 44.

Knowledge of the material in this chapter is not necessary in order to use builtin scripts provided with EMTWorks. It is for those users who would like to modify existing scripts and to create new scripts for customization purposes.

In addition to the Export script language, EMTWorks provides JavaScript based scripting. Scripts created using the Export script language can be also called from JavaScript based scripts using the “runExportToString” command documented in the EMTWorks electronic manual.

1.1. Command Language Introduction

The script language implement by the Export tool was developed originally as a flexible way of specifying Netlist formats; an alternative to the hard-coded Netlists that are provided by most schematic packages. However, as enterprising users pushed the limits of the original language, we added more features to support more complex report formats, error checking, back annotation and other applications. The script language is still primarily oriented around the concept of generating a text file from design data. However, the new regular expression and file I/O features added in this version make many other applications possible.

1.1.1 Basic Script Structure

A script file is simply a text file containing any number of lines of text with optional commands and comments embedded in them. The default mode of operation is to read the input (script) file one line at a time, scanning for commands. If no commands are detected, the line is simply written directly to the output file. In fact, the program will accept any text file as input. If it doesn't recognize any commands, the entire file will be written verbatim to the output without modification (except, possibly, for a change in line terminators - more on this topic later).

When reading the script file, the program scans for any commands, variables or comments. If any of them are found, new data may be generated and substituted at that location in the output, or some other action may be taken.

COMMANDS – Commands start with the “\$” symbol and are in upper case letters. Commands are used to control the format of the report and perform various actions. Some of the commands need arguments to set certain values (e.g. the number of characters on a line). Any arguments are enclosed in parentheses following the command keyword.

See more information on command argument format in “Command Arguments” on page 15.

If the desired output file format requires that a word starting with a “\$” be placed in the script, it is best to precede the “\$” with the escape character “\” (backslash) to ensure that it will not be accidentally interpreted as a command.

Commands fall into the following general sub-groups.

Definition commands are those which set parameters for how data is generated but do not themselves generate any text in the output file. An example is the \$SIGPINFORMAT command, which specifies how pins will be displayed in a signal list.

Action commands cause some action to be taken but don't generate any text data that ends up in the output file. For example the \$SORT command sorts a device or signal list.

Function commands perform arithmetic, logical and various conversion operations on data supplied as arguments. For example \$PLUS will add two numbers and return the result.

Listing commands are those which cause lists of items to be directed to the output file. For example the \$SIGNALS command, which generates a list of the signals in the circuit. All the commands which follow a listing command on a line are item-specific, that is, their exact meaning depends upon the type of item being listed.

Object data commands are those whose value derives from the circuit object being listed. These commands can be context-sensitive and may not have meaning in all locations in a script. For example, the \$SIGNALNAME command can appear in a \$SIGNALS listing and causes the name of that signal to be substituted. The \$SIGNALNAME keyword has no meaning standing alone in the script.

System data commands cause a specific internal value to be directed to the output file, e.g. the design file name or the system date.

ATTRIBUTE OR VARIABLE REFERENCES – References to script variables or device, pin, signal or design attributes start with the "&" symbol. The reference will be replaced with the actual value of the attribute or variable. More information on variables can be found under "Script Variables" on page 17.

If the desired output file format requires that a "&" be placed in the script, it is best to precede the "&" with the escape character "\" (backslash) to ensure that it will not be accidentally interpreted as an attribute reference.

COMMENTS – Comments are text enclosed between left and right braces "{" and "}". Comments are intended for internal documentation of the command file. No characters between, including the braces or a carriage return following the closing brace, will be transmitted to the report. If you need to transmit the character "{" to the output file, you must precede it by the "escape" character "\" (backslash) so that it will not be recognized as a comment.

All other text and special characters (e.g. tabs and form feeds) found in the command file are considered to be raw text and are transmitted to the output file unmodified.

Any item starting with a \$ or & but not recognized as a command or variable word will be transmitted to the report file. No error messages are generated for incorrect keywords.

It is recommended that only normal ASCII characters be used in reports. Using special symbol characters (e.g. using the Character Map tool in Windows) may result in difficulty transmitting the files to other programs and some of the Report format options may not function correctly.

A number of standard script files for report generation and error checking are provided with EMTWorks and some of these are explained in detail in "Script Examples" on page 44. These can be used as guides in creating your own scripts.

I.1.2 EMTPWorks Object Types

For report generation purposes, an EMTPWorks circuit consists of two major types of objects: devices and signals. The script tool has two general sets of options allowing information on each of these types of objects to be extracted or written to a file in a variety of forms.

Devices

The device listing features are used to create component lists, bills of materials and types of Netlists that are listed by device, such as a SPICE or EMTP format. For each device in a list, any of the following data can be included:

Device Name—this is the name applied to the device instance, e.g. “U1”, “R23”, “J12”, etc.

Device Attributes—the contents of any device attribute fields can be included in a listing.

Device Token—the device token is an integer that is assigned automatically by EMTPWorks when the device is first placed on the diagram and not changed as long as that device exists in that circuit. This token number can be used by an external program to determine which devices are new and which are old in a circuit, despite name or parameter changes.

Device Sequence Number—this is a temporary integer that is assigned by the Report tool as part of its sorting process and may change during a report if multiple sorts are done. This can be used in cases where external programs require objects to be sequentially numbered.

Device Type information—this refers to information retrieved from the library when the device was created, such as the type name (e.g. “C grounded”), number and types of pins, etc.

Device Graphical Information—information about the device's page number, position and orientation on the schematic is available to allow transfer of schematic data to other systems, or to create reference listings.

Attached signals—a device listing can also contain a list of the signals attached to each pin of the device. Each signal entry can reference any of the signal data described below. See “\$DEVPINFORMAT”.

Device listings are created using the \$DEVICES command.

Signals

Signal listings are used primarily for creating Netlists for transfer to external simulators or PCB (Printed Circuit Board) systems. The following information is available about each signal in the circuit:

Signal Name—this is the name of the signal as it appears on the schematic, or is assigned automatically by the report generator.

Signal Attributes—the contents of any signal attribute field can be included in a signal listing.

Signal Token—this is an integer that is assigned automatically by EMTPWorks when the signal is first created and not changed as long as that signal exists in that circuit. Note however, that many common editing operations result in signals being merged or broken apart, which will often result in the signal token changing even though some of the connections in the old signal remain. This token number can be used by an external program to determine which signals are new and which are old in a circuit, despite name or parameter changes.

Signal Sequence Number—this is a temporary integer that is assigned by the Report tool as part of its sorting process and may change during a report if multiple sorts are done. This can be used in cases where external programs require objects to be sequentially numbered.

Attached device pins—any signal list can contain a list of the device pins interconnected by this signal. Pin format options allow almost any of the device parameters mentioned above to be included in each pin entry. See “\$SIGPINFORMAT”.

Signals listings are created using the \$SIGNALS command described later in this chapter.

I.1.3 Definition Commands

Definition commands control the general format of the report, for example: the number of rows per page, the number of columns per row and other report formatting options.

Each of these parameters is set by using specific command keywords, the most common of which are summarized in the following table.

Complete descriptions of these commands is provided in “Part II – Script Keyword Reference” on page 51.

\$LINEWIDTH(<i>n</i>)	This command specifies the maximum number of characters that will be placed on a line before an automatic carriage return is inserted.
\$PAGELENGTH(<i>n</i>)	Specifies the action to be taken when a page is full (i.e. when the number of lines specified in \$LINESUSED is exceeded).
\$LINESUSED(<i>n</i>)	Specifies number of lines to use on each report page before moving to the next page.
\$MAXITEMSPERLINE(<i>n</i>)	Restricts the number of entries placed on a line in repeating structures such as a netlist.
\$AUTONUMBER(<i>number of pins</i>)	Any device with less than or equal to the number of pins specified will have pin numbers automatically assigned if none were present in the circuit file. This option is intended to provide pin numbers for discrete components in a circuit, since they do not normally have pin numbers on a diagram. The default number is zero.
\$DEVPINSEQUENCE & <i>attrField</i>	Specifies that the order in which pins appear in a pin listing is to be determined by an attribute specified in the device. This is intended for generating report formats such as SPICE, in which pin order is significant, when some devices in a design may not have their pins defined in the required order.
\$COMBPINSON/\$COMBPINSOFF	When ON causes multiple pin connections on the same device to be combined. Default is OFF.
\$ALIGNCOLSON/\$ALIGNCOLSOFF	When ON causes extra blanks to be inserted between netlist or component list entries to force column alignment.
\$SPACE(<i>number</i>)	Sets the column spacing used when \$ALIGNCOLSON/\$ALIGNCOLSOFF is set. Default is 16.

\$ITEMSEPARATOR(<i>string</i>)	Specifies a string that will be inserted between successive items in an item list.
\$UNNAMEDDEVS(<i>string</i>)	Specifies a string that will be output whenever a device name is called for and the device in question is unnamed.
\$UNNAMEDSIGS(<i>string</i>)	Specifies a string that will be output whenever a signal name is called for and the signal in question is unnamed. The default value is "unnamed".
\$ONEPINSON/ONEPINSOFF	When ON suppresses the pin number from being printed in netlist entries for devices with only one pin. This will cause entries such as test points to appear as TP1 instead of TP1-1. Default is OFF.
\$CONTSTART(<i>string</i>)	Specifies a string of characters that will be inserted at the beginning of each line that is a continuation from a previous line (i.e. whenever \$MAXITEMSPERLINE or \$LINEWIDTH is exceeded while writing a list).
\$CONTEND(<i>string</i>)	Specifies a string of characters that will be inserted at the end of each line that is being continued onto the next line (i.e. whenever \$MAXITEMSPERLINE or \$LINEWIDTH is exceeded).
\$BLANKREPLACE(<i>string</i>)	Specifies a string of characters that will be substituted for a blank in any device or signal name. This is done to accommodate systems which cannot accept blanks in names.
\$UNCONNPINSOFF/\$UNCONNPINSON	When ON, allows device pins that have no signal lines attached to them and are not connected to any other signals by name to appear in the netlist, otherwise they are suppressed. Default is ON.

This section provides information on a number of concepts that are important to the operation of scripts.

I.1.4 Current Design or Current Object

A script can be invoked under many different circumstances in addition to the simple Export command in the File menu. A script can sometimes be executed in response to an action on a specific object, such as a device. Similarly, one part of a script may cause another part of a script to be invoked to generate data for a specific object.

For these reasons, it is important to be aware of what the “current object” is at any point in a script. In fact, normally there is a sort of hierarchy of objects that may be involved in evaluating a given command. For example, suppose you are generating a listing for a pin in a netlist. The pin is associated with a parent device and an attached signal, which are in turn associated with the parent circuit, which is in turn contained in some design. Data from any of these objects can be used in generating a pin listing.

It is important that a script not attempt to access data for a “smaller” object than the current one. E.g. If the current object is a device, you cannot use the \$PINNUM keyword because it requires access to a specific pin. In general, a device has many pins and the Export tool cannot assume that it knows which one you mean. On the other hand, if the current object is a pin, you can use \$DEVNAME to get the attached device’s name, because there is an unequivocal device associated with a pin.

The following points summarize how the current object is determined.

If a script is executed by a user command from a menu, the current object at the start is the current circuit (i.e. the topmost circuit window).

If a script is invoked as a result of some operation on an object (e.g. placing a device), then the current object will be the one just edited (i.e. the device just placed).

Any part of a script that is executed as part of an iterating command (like \$DEVICES) will have as its current object the current one in the iterating sequence.

I.1.5 Data Types

The script language only really implements one type of data: the character string. The internal representation of character strings limits their length to 32,767 characters. This does not limit the total length of an output file, but no single line, variable value, or attribute value can exceed that length. Any commands that require other types of data (e.g. numeric or Boolean) will convert the arguments from the character string format to the required internal format, then convert the results back to a character string again.

Integers

Some of the keywords expect values that would be considered to be an “integer” data type in a regular programming language. For commands such as \$PLUS, the arguments will be assumed to be the string representation of decimal integers and will be converted to signed 32-bit internal values for the operation. The result is then converted back to a decimal integer string.

No errors are reported during this conversion process. Conversion of a string to an integer stops as soon as any non-decimal character is encountered. Thus, an empty string, or any string starting with a non-numeric character, will be treated as zero.

Booleans

Some of the commands operate on or return “Boolean” (i.e. TRUE or FALSE) values. Again, the internal representation is a character string. For commands such as \$IF that evaluate the “truth” or “falseness” of the data, an argument will be taken as TRUE if it contains any non-zero-length value. Note that the contents of the string are not examined, so even a blank character will count as TRUE. An argument will count as FALSE only if it is zero length. (Note that there is a \$NONBLANK function which can be used if you want to count a “blank” string as FALSE.)

For commands that return a “boolean” value, such as \$AND, TRUE is represented by a string consisting of a single character “1”. FALSE is represented by an empty string.

I.1.6 Blocks

A number of script commands refer to a block. A block is simply a set of lines in the script file that start with a \$DEFINEBLOCK command and end with an \$END. Once defined, a block is treated internally as if it was a separate file. When the script is executed, everything between the \$DEFINEBLOCK and corresponding \$END is skipped until some other part of the script makes explicit reference to it.

Blocks have a number of uses.

They can be used as a kind of “subroutine” which can be invoked from elsewhere in the script using the \$INCLUDE command. This can be useful if several parts of a script need to make use of the same sequence of lines.

They can be used simply to delimit a block of text to define a value mapping table for the \$TABLE, \$MAP or \$VERIFY commands.

They can be used to define a set of script lines to be executed under specific circumstances, e.g. to define the format of an internal circuit in a hierarchical netlist, as in the \$INTERNAL command.

The \$DEFINEBLOCK command is an extension of the \$DEFINECIRCUIT command used in hierarchical report forms.

I.1.7 Command Arguments

Many \$ commands can be followed by argument strings contained in parentheses. These arguments can pass information used in performing the requested operation. In some cases the arguments are optional and the command will perform a default function with no arguments. For example, when the \$DATE command is used without following argument in parentheses, it returns a string containing the date of the report in a standard format. If you want to control the format, you can add a string argument containing format control characters.

Argument Data Types

As discussed in “Data Types” on page 14, there is really only one kind of data in the Export tool, the character string, but it may be interpreted in various ways by different commands. Various kinds of arguments are used, depending on the command:

String—everything is a string.

Integer—this is the string representation of a decimal integer.

Boolean—this is a string in which any non-null value will be taken as TRUE and an empty string is taken as FALSE.

Block—some commands refer to another block of text elsewhere in the command file. Such a block starts with a \$DEFINEBLOCK command (which specifies the name of the block) and ends with an \$END. The block can then be referred to by name by various commands. Blocks are used to specify value mapping tables, sets of commands to execute under specific circumstances, and other uses.

Argument Syntax

For any command that takes arguments, the argument list is enclosed in parentheses and the individual arguments are separated by commas. Because the arguments are literal strings where every character is potentially significant, several special cautions should be observed:

Leading and trailing white space is removed from argument strings. If you want to pass a leading or trailing blank or tab in an argument string, or a single blank by itself, you must escape it with a backslash. For example, the following sets the variable to a string consisting of one blank:

```
$SETVAR(Blank, \ )
```

If you need to pass a comma or closing parenthesis in an argument string, these should also be escaped, as in:

```
$TABLE(ValueList, \,, \))
```

Note that this is passing three arguments. The first is a block name, the second is a string containing a comma and the third is a string containing a closing parenthesis.

I.1.8 Control and Escape Characters

Anywhere in a script file where literal text is allowed (i.e. anywhere not inside a \$ command structure or comment), the following two format options can be used to insert special characters in the output stream:

Control characters (i.e. ASCII codes less than 20 hex) can be inserted in the file using the ^ (shift 6) character. The character following the ^ will be inserted in the file with 40 hex subtracted from its ASCII code. This means that ^@ will generate a null (code 00) character, ^A will generate code 01, ^M will generate 0D (carriage return), etc. Note that the letter following ^ must be upper case.

Special characters that may have specific meanings in the report command language can be included in the output using the \ escape character. For example, the brace character { is used to open a comment block in the command language. To include this character in the output stream, it should be preceded by the escape character, as in “\{”. In addition, the escape character can be

used to terminate keywords or attribute field names that might otherwise run into the following literal text. For example:

```
&ExtraText\ENDMODULE;
```

In this case the \ is used to separate the field name “ExtraText” from the literal text “ENDMODULE;”. Without the \ escape character, the command language interpreter would consider the text ExtraTextENDMODULE to be an attribute field name.

I.1.9 Script Variables

A script can create, modify and refer to string variables. The following notes outline the rules for string variables:

A variable is defined and set by the \$SETVAR command.

A variable is referred to (i.e. its value is extracted) by using the variable name prefixed by an ampersand “&”. Note that this is the same way that an attribute field in the design is referred to, so it is the user’s responsibility to ensure that there is no accidental name overlap. If you create a variable with the same name as an attribute field, the variable will take precedence. I.e. A reference to &name will return the value of the variable.

A variable is “global” throughout the script. I.e. once a variable of a given name is set anywhere, its value remains set and it can be referred to in any part of the script executed later.

Variable values disappear when the script finishes executing, i.e. if you run the same or another script later, the same variable will not retain its old value.

Variable names can be 1 to 16 characters long and can consist of letters, numbers, dots “.”, and underscores “_”. Variable names consisting entirely of decimal digits are reserved, as discussed below.

The following simple example shows how a variable is set and referenced:

```
$SETVAR(MyVar, This stuff becomes the value)  
The value of the variable is: &MyVar
```

Special Variables &1—&9

Variable names consisting entirely of decimal digits are reserved for use as regular expression “match” variables. When a regular expression is executed, these are automatically set to the values matched by parts of the expression. A complete description of this facility is given in the section “Regular Expressions”.

Predefined Variables

You can predefine variables and set them a certain value by using the SCRIPTVAR keyword in the .INI file described in “Using EMTP”. This can be useful to allow you to make a script that uses variables for items that may change from user to user, such as directory pathnames. A path variable can be defined once in a setup file and a setup of scripts can then refer to them. The variable can then be changed by changing only one reference in the Setup file and all scripts will then use the new value.

Note that these variables are not “global” in that they are reset to their defined values at the beginning of each of each script. You can modify them using the \$SETVAR command, but they do not retain the modified value when the same or any other script is executed. If this type of behaviour is needed, it can be implemented by storing values in attribute fields associated with the design.

I.1.10 Attribute Field References

The ability to easily extract the values of device, signal, pin or design attributes in a report is one of the most important features of the script language. An attribute field is referred to (that is, its value is extracted) simply by using the attribute field name prefixed by an ampersand “&”. For example, the following single line script will list all the devices in a circuit, showing the name followed by the contents of the Part and Value fields:

```
$DEVICES$DEVNAME &Part &Value
```

The object type that is being referred to by a given attribute reference depends on its position in the file and on how the script was invoked. With the exception of pins (described in the next section), an attribute reference always refers to the current object. In most circumstances, this means that a field reference that appears in a \$DEVICES line refers to a device attribute, a reference that appears in a \$SIGNALS line refers to a signal attribute and a reference that appears in a top-level line (i.e. not in \$DEVICES or \$SIGNALS) refers to a design attribute.

See the rules for determining the current object in “Current Design or Current Object” on page 14.

I.1.11 Precedence of Field References in Pin Listings

Some ambiguity can arise in attribute references when the current object is a pin, that is, inside a \$PINS command. In order to allow reference to attributes belonging to the pin itself or the associated device or signal, the Export tool searches for values in the following order:

If the named field has a non-null value in the current pin, it is used.

Otherwise, if the field has a non-null value in the device associated with the current pin, it is used.

Otherwise, if the field has a non-null value in the signal associated with the current pin, it is used.

This allows you to specify a default value for an attribute in a device or signal and then override it for selected pins. For example, the following script refers to an attribute field “ConnType” that has been defined for both pins and devices:

```
$DEVPINFORMAT ($PINNAME=&ConnType)  
$DEVICES$DEVNAME $PINS
```

If the field ConnType has a value of “Normal” in the device, a value of “Special” in the third pin and a null value in all other pins, you will see a listing like the following:

```
DEV1 PIN1=Normal PIN2=Normal PIN3=Special PIN4=Normal
```

I.1.12 Attribute vs. Variable References

Note that attributes are extracted the same way as script variables, so it is the user's responsibility to ensure that there is no accidental name overlap. If you create a variable with the same name as an attribute field, the variable will take precedence. That is, a reference to &name will return the value of the variable.

When the script language parser is scanning for commands, it takes any contiguous group of identifier characters as a single identifier. This can be a problem if you wish to append some fixed text to the extracted value from a field. For example, if you wished to append “_000” to every part number (from a field called PartNum) in a bill of materials, it might be tempting to put this in a script:

```
&PartNum_000
```

However, the script language will interpret this as a reference to a field called PartNum_000, since all these characters are valid in an identifier. The solution is to insert an escape character after the identifier to clearly delimit it, as in:

```
&PartNum\_000
```

1.2. Controlling Report Page Layout

A number of script keywords and features are designed to assist in creating neat, human-readable reports. This section will cover the setting of page height and width, page and value breaks and column alignment.

Specific details on each of the keywords used here are provided in “Part II – Script Keyword Reference” on page 51.

1.2.1 Setting Page Height and Width

Three keywords control the number of lines of text output on each page and the numbers of characters on each line: \$LINEWIDTH, \$LINESUSED and \$PAGELENGTH.

The Export tool is generating a pure text file and has no direct control over the fonts, tab settings, margins and other page layout parameters used when printing the report output. It is the user's responsibility to understand the relationship between these settings and the actual line and character counts used in generating the reports.

A typical usage of these commands would appear as:

```
$LINEWIDTH (60)
$LINESUSED (56)
$PAGELENGTH (66)
```

This indicates to the program that the maximum number of characters to be output on a line is 60 and the maximum numbers of lines of text on a page is to be 56. Note that the Export tool does not break individual text items on output. The line width is only used to decide whether or not to put out the next item generated by a repeating data command. If the fixed content of a line (i.e. outside of repeating data items) is greater than the specified width, it will be output regardless.

The \$PAGELENGTH command indicates the total number of lines on a page, including the \$LINESUSED, the page header and any blank lines generated to fill out the page. In this example, the total page length is 66 lines. If no header has been specified, once the 56 lines specified by \$LINESUSED have been generated, the program will generate 10 empty lines to meet the specified total. Obviously, the number of lines that will fit on a page depend on the font and line spacing settings. It may be necessary to experiment to choose an appropriate \$PAGELENGTH setting for your output medium.

A special \$PAGELENGTH value of zero can be used to indicate that an ASCII form feed character should be output to indicate a page eject. Many word processing packages accept this character as a page break.

1.2.2 Defining a Page Header

Any block of text can be defined as a page header to be inserted at the top of each generated page. This text is defined in a section that starts with a \$HEADER keyword and ends with an \$END keyword. The contents of this section is passed to the output file immediately to create the header for the first page, then each time a page feed occurs.

The header section is scanned for any commands or variables to substitute before it is output. Any command keywords that are shown in the keyword reference as valid in “Script” or “Script - Design” can be used in the header section. These allow you to insert the page number (\$PAGE), date (\$DATE/\$TIME), design name (\$DESIGNNAME/\$FILENAME), contents of design attributes (&field) and other values in the header section.

Here is a simple example of a page header specification:

```
$HEADER
XYZ Industries of Smithville, Inc.
Bill of Materials Report
File: $FILENAME    Date: $DATE
$END
```

I.2.3 Setting Column Alignment

Since the Export tool is generating a text file and does not print directly, the actual column alignment that is obtained on output depends on the mechanism used to print the final report. There are two general ways of ensuring alignment of columns of text in reports generated by the Export tool:

You can use the \$COL command for fixed items or the \$ALIGNCOLSON/\$ALIGNCOLSOFF command for repeating items. This inserts extra blank characters in the output to ensure column alignment. This only provides correct alignment on output when used with a fixed-space font such as Courier.

You can insert tab characters using the \$TAB and \$ITEMSEPARATOR commands and then set the tab spacing as desired in the word processor you use to print the results.

I.2.4 Defining a Value Break

The Export tool allows you to define a block of text to be output each time a computed value changes while generating a sorted \$DEVICES or \$SIGNALS listing. This allows you to insert a new heading or simply a blank line for extra readability between sections of a report. For a complete description of this feature, see command \$BREAK.

1.3. Date and Time References

The script language provides a method for displaying and manipulating date and time values in a variety of formats. In addition to the current date and time provided by the operating system, date and time values are used in a number of EMTPWorks operations, including:

The created and last modified dates of the current design file are available for use in reports. Note that “last modified” refers to the disk file associated with the design, not the design itself. Schematic editing operations do not change this value. For designs that have not yet been saved to a file, these will be the date the file was created.

The last modified date of the source device library is stored in a device attribute called LibDate when a device is placed in a design. This can be used later to determine if the design is up to date with respect to the symbol library.

The last modified date of the external circuit file is stored in a device attribute ExtCctDate when the external circuit is attached as a sub-circuit.

When a device is placed in a schematic, the current time is stored in the DateStamp.Dev field of the device. This can be used in back- and forward-annotation processes that need to uniquely identify devices even if the name has been changed.

See more information on date stamps in the chapter entitled “Device Naming and Packaging” in the EMTPWorks User’s Guide.

1.3.1 Raw Date and Time Format

In EMTPWorks date can be stored automatically in an attribute field. This is an integer value that represents the number of seconds since January 1, 1970. When this value is stored in an attribute, it is converted to an unsigned decimal character string. The \$DATE and \$TIME commands provide formatting operations to convert these values to human-readable form.

1.3.2 Date and Time Formatting Commands

The \$DATE and \$TIME commands can be used in a number of different forms to display the current system time or any stored time value used in EMTPWorks time-stamping functions. The \$DATE and \$TIME commands are described in detail in “Part II – Script Keyword Reference” on page 51.

I.4. *Sorting and Merging*

The script language provides the ability to sort device and signal listings on any data field. In addition to the obvious application of producing sorted listings for improved readability, sorting provides the basis for these additional functions:

A merging function allows all devices or signals with the same sort value to be combined into a single line in the output. This can be used, for example, to create a “bill of materials” style listing in which a single line is devoted to one part number and lists all instances of that part number. This is described in more detail in “Enabling Merging” on page 25.

A value break function, \$BREAK, allows the insertion of a header or page break when some field or computed value changes.

I.4.1 The \$SORT Command

The \$SORT command provides the ability to sort device and signal listings on any data field. Items with the same value in any field can be optionally merged into a single line. This allows listings to be organized to suit various applications, for example:

Device or signal lists can be sorted by name to enhance readability.

Device lists can be sorted by type so that each line lists one device type with all information common to that type and a list of instances of the type.

A device list can be sorted by page number to show the devices used on each page.

Signals can be sorted by an attribute field, for example to give priority to certain nets so they are listed first for autorouting purposes.

Devices can be sorted by any attribute field, for example by component value, stock number, cost, etc.

The \$SORT command has the following form:

```
$SORT objectType field1 field2 ...
```

where:

objectType must be either “\$DEVICES” or “\$SIGNALS”.

field1, *field2*, etc. are identifiers indicating which fields to sort on. The list is sorted first on the first field. If any items in the list have identical values in that field, then the groups of like-valued items are sorted on the next specified field, etc.

Once a sort has been done, it remains in effect for all subsequent listings on that object type until the next \$SORT or \$FIND command. Each \$SORT clears the previous \$SORT. Sorting can be in ascending or descending order, as described below.

The following field specifications are valid in all \$SORT commands:

\$ASCENDING	This specifies that all following fields specified should be sorted in ascending order. This is the default direction, so ASCENDING will normally only be required after a
-------------	--

	DESCENDING to reset the sort direction for subsequent fields on the same line.
\$DESCENDING	This specifies that all following fields will be used to sort in descending order.

If *objectType* is \$DEVICES, the following field specifications can be used:

\$DEVNAME	Sort on the device name. If no name exists in the circuit, and no name has been assigned using \$ASSIGNNAMES, then the item will be placed at the end of the list. If the device name contains a numeric portion (e.g. U12), then the numeric part of the name is sorted on its integer value, instead of its character value. E.g. the name U12 will appear before U110, instead of after, as it would in a purely lexical sort.
\$DEVHIERNAME	Sort on the device hierarchical name, i.e. a name generated by prefixing the device name with the names of all parent devices in the hierarchy. This is normally only used in flattened netlists.
\$DEVINSTNAME	Sort on the contents of the InstName field. This is normally only used in flattened netlists.
\$POSX	Sort on the horizontal position of the top left corner of the device symbol on the diagram.
\$POSY	Sort on the vertical position of the top left corner of the device symbol on the diagram.
\$TYPENAME	Sort on the device's type name, i.e. as it appears in the Parts palette.
\$ORIENT	Sort on the device orientation.
\$PAGE	Sort on the page number that the device appears on.
\$NUMPINS	Sort on the number of device pins.
\$NUMEMTPPINS	Sort on the real number of device pins
\$DEPTH	Sort on the contents of the Depth attribute field. NOTE: This field will only be valid if the mode is enabled.
&attrField	Sort on the contents of the attribute field named attrField in each device. If that field is empty or does not exist in a given device, then that device will be placed at the end of the list.

If *objectType* is \$SIGNALS, the following field specifications can be used:

\$SIGNALNAME	Sort on the signal name. If no name exists in the circuit, and no name has been assigned using ASSIGNNAMES, then the name "unnamed" will be assumed. See the note under "DEVNAME" above regarding sorting numeric portions of the name.
\$SIGHIERNAME	Sort on the signal hierarchical name, i.e. the name of the signal prefixed by the names of all parent devices in the hierarchy. This is normally only used in flattened netlists.
\$NUMPINS	Sort on the number of device pins attached to this signal.
\$SIGSOURCE	Sort on the order of appearance of this signal in a SIGSOURCE command. If the signal does not appear in a SIGSOURCE command (as most won't), it will appear after the SIGSOURCE signals. This is used to place all power and ground nets at the front of a netlist.
&attrField	Sort on the contents of the attribute field named "attrField" in each signal. If that field is empty or does not exist in a given signal, then that signal will be placed at the end of the list.

See the examples in “Script Examples” on page 44 for more information on how to use the sorting capability.

I.4.2 Enabling Merging

The combination options \$COMBSIGSON/\$COMBSIGSOFF and \$COMBDEVSON/\$COMBDEVSOFF instruct the Export tool to combine items with the same sort value into a single line in the listing. This can be used to produce a variety of useful listing formats, for example listings of all devices having a certain type name.

The “COMB” options will have no effect if no sort has been specified since they only operate on sorted values.

\$COMBSIGSON/\$COMBSIGSOFF When ON causes signals with the same sort value to be combined into a single net in the netlist. Default is OFF.

\$COMBSIGSON/\$COMBSIGSOFF should not be used in most normal netlisting applications. Any like-named signals have already been logically connected by EMTPWorks and will be listed in one entry even if COMBSIGS is OFF. COMBSIGSON will have the additional effect of merging like-named signals, even if they appeared on different pages of the schematic.

\$COMBDEVSON/\$COMBDEVSOFF When ON causes devices with the same sort value to be merged into a single entry in the component list. Default is OFF.

Display of Multiple Values in Merged Lines

Whenever merging is enabled (that is, \$COMBSIGSON/\$COMBSIGSOFF or \$COMBDEVSON/\$COMBDEVSOFF is active), a single line in a listing can represent an arbitrary number of circuit objects. All objects on that line have the same sort value but will generally differ in other values.

For example, this simple script:

```
$SORT $DEVICES &Part
$COMBDEVSON
$DEVICES&Part $DEVNAME
```

It will produce a sorted list with one line per Part value, sorted by Part. When the Export tool encounters \$DEVNAME it actually puts out the names of all the devices merged on that line, so a typical line will look like:

```
YgDp30 YgD_1 YgD_2 YgD_3
```

Items with multiple values will always be sorted by value and are always merged, i.e. only unique values are displayed. In the example above, if multiple device symbols have the same name, that name will only appear once in the list.

Disabling Multiple Value Display

There may be cases where the display of multiple values on merged lines would not be desirable. An example is the use of date stamps in PCB netlist output. If multiple gate symbols are merged into a single package in the netlist, the date stamp value will be different on each of the device symbols being combined into one package. However, the PCB package only wants to see a single value since it only considers it to be one object. There are two ways of creating a single value for use in the report.

The `$SINGLE` keyword disables multiple value display for the next command that generates any object-derived data. When this is used the value that will be displayed in the first one encountered in the internal circuit structure. There are no guarantees that this value will be the first one in any sort sequence.

You can use a function call to derive a value from the list of object values. There is a built-in function that may be useful for this: `$COUNT` (a count of the number of items).

Multiple Value Items in Function Arguments

When a data item that represents multiple values is used as an argument to a function command, the question arises of whether the value is expanded before or after the function is called. The Export tool's default behaviour is to expand multiple values after a function is evaluated. Put another way, data items that are passed as arguments to a function are not expanded to multiple values. It is assumed that the function will be called once for each object being merged and that the data item in the argument will take on its appropriate value for that single object.

This default behaviour can be overridden using the `$MERGE` keyword, which can be thought of as the opposite of `$SINGLE`. If `$MERGE` is placed before a data item, its value will be the expanded value of all objects that it represents even if it is an argument to a function.

We will illustrate this by a simple example. Suppose we have a circuit with two devices having the same value in the Part attribute, but different values, "1" and "2" respectively, in the Value attribute. We then execute this script:

```
$SORT $DEVICES &Part
$COMBDEVSON
$SETVAR(Test1, )
$SETVAR(Test2, )
$DEVICES&Part $SETVAR(Test1, &Value) $SETVAR(Test2, $MERGE&Value)
The value of Test1 is &Test1
The value of Test2 is &Test2
```

This script will generate the following output:

```
The value of Test1 is 2
The value of Test2 is 1 2
```

In this case the first `$SETVAR` command is executed twice, once with Value equal to "1" and once with Value equal to "2". The second call overwrites the value of Test1 set by the first call. Note that there is no guarantee of the order of execution, so the value output might be "1".

The second `$SETVAR` has a `$MERGE` command, indicating that we want the `&Value` item to be substituted with all the values of the merged objects, in this case "1 2".

The second \$SETVAR is still executed twice, even though it produces that same value both times. This could be overridden by placing a \$SINGLE in front of \$SETVAR. In this case, the result would not be affected, but it would be a minor inefficiency.

1.5. Implementing Mark as OK in Error Checking Scripts

A number of commands are provided to assist in creating a “Mark as OK” function in error checking scripts. This is primarily intended for use with the ErrorScript tool (Find in the Drawing menu), although it could be used elsewhere. The ErrorScript tool has a “Mark as OK” button which the user clicks to indicate that the current object found by the error checking script should be considered to be not in error and not be located again by future runs of the same error check. It is the error checking script’s responsibility to implement this function and to provide the necessary data to the ErrorScript tool.

1.5.1 How the Mark as OK Function Works

In order to allow a “Mark as OK” setting to be specific to a certain test, a value is stored in the OKErrors attribute field which represents a bit set numbered from 0 to 31. In designing a set of error checks for a design kit, a bit number is assigned to each error check. When it is scanning circuit objects looking for errors, the script should check its assigned error bit. If the bit is on, then this error check has been masked out for this object and the object should not be considered to be in error.

In order to reduce duplicate usage of error bits, bit number 0 to 15 are reserved for use in scripts provided with EMTPWorks. Users are encouraged to use bits 16 to 31 for their own scripts. Note however that bits number 17, 18, 19, 21, 22, 23 and 25 are presently already used by the script located in file EMTP Error Check.rfm.

1.5.2 Error Bit Functions

This table summarizes the functions available to operate on the contents of the OKErrors field:

<code>\$ERRORBITON(num)</code>	This returns a TRUE value if the given bit number is 1 in OKErrors, that is, if the corresponding error check should be skipped for this object.
<code>\$ERRORBITOFF(num)</code>	This returns a TRUE value if the given bit number is 0 in OKErrors.
<code>\$SETERRORBIT(num)</code>	This sets the given bit number in OKErrors, i.e. marks this error as OK.
<code>\$CLEARERRORBIT(num)</code>	This clears the given bit number in OKErrors.
<code>\$CLEARERRORS</code>	This sets the OKErrors field to a null value, i.e. re-enabling all error checks for this object.

There is no absolute requirement that you use the OKErrors attribute field or any of the functions described here in creating an error checking script. You are free to use any other attribute fields or script functions in implementing error checks. However, unless there is a good reason for doing otherwise, we suggest using these functions so that the user of your scripts sees a consistent implementation.

I.5.3 How the Mark as OK Value is Stored

The “Mark as OK” bits are stored as an unsigned 32-bit integer represented in hexadecimal form in the OKErrors attribute field. This field is predefined as a secondary field for devices, pins and signals in all designs created in EMTPWorks.

I.6. Reporting Power and Ground Nets

The “signal source” facility in the Export tool allows you to name certain signals to be treated as power and ground nets. This has the following two effects:

It allows you to create common connections in a circuit using attribute entries in devices. This can be used to create power and ground entries in the Netlist without having to show all these connections explicitly on the diagram.

It informs the Export tool that the given signals should be merged across hierarchy levels.

For an overview of the methods of creating power and ground connections, see “Using EMTP”.

I.6.1 Specifying Signal Sources

Any device attribute field may be specified as a signal source by the command:

```
$SIGSOURCE(signalName) &fieldName
```

signalName is the name of the signal to attach pins to in the netlist. *fieldName* is the name of the attribute field to search for in each device to look for pin numbers to attach to the named signal. If *fieldName* is omitted, the signal name is taken to be the name of the attribute field to search.

The named device attribute field is assumed to contain a list of pin numbers to attach to the signal. This can consist of a single pin entry, such as “7”, or a list, such as “5,6,9,14”, separated by commas.

In this version of EMTPWorks, pin numbers are limited to 4 characters. If you omit the comma in a list like the one above, the entire text string will be taken as a single item and the Export tool will report a string overflow error.

For example “Ground” connections can be created as follows.

Using the Attributes command on a selected device, create an entry such as “7” in the Ground attribute for a device, in this case connecting pin 7 to the Ground net.

Place the command `$SIGSOURCE(Ground)` in the script file. This causes Export tool to search all device attributes for fields named Ground and use the field value as a pin number.

There is no fixed limit to the number of SIGSOURCE entries that can be created.

The predefined fields Ground and Power are normally used for standard power and ground connections. These pin connections are prespecified for all digital components in the standard EMTPWorks libraries. Corresponding \$SIGSOURCE statements are included, where appropriate, in all standard netlisting scripts. You can create your own special-purpose power nets by using the \$DESIGNPINSIGSOURCE/\$DESIGNSIGSOURCE facility described in the following section.

I.6.2 Creating Design-Specific Signal Source Fields

Including a \$SIGSOURCE command in a script creates a signal source that will be searched for all designs that use that script for output. In some cases it is desirable to specify a special power

or ground net that applies only to a single design. For this purpose, most of the standard report scripts included with EMTPWorks have a line like this:

```
$DESIGNSIGSOURCE (SigSources)
```

This command provides a more indirect and flexible way of specifying signal source fields independently for a single design.

The *SigSources* design attribute field is predefined in all designs for this purpose.

This command causes the Export tool to take the following steps:

The attribute field named as an argument of the \$DESIGNPINSIGSOURCE/\$DESIGNSIGSOURCE command is taken to be a design attribute field. This field is retrieved and scanned for a list of signal source names like this example:

```
Minus5V, Plus3V
```

Each of the items in this list is treated as a signal source. That is the named field is checked in each device for a list of pin numbers and any found are added to a net with the same name.

In this case, placing “Minus5V, Plus3V” in the design field SigSources is equivalent to placing the following lines in the script, for this design only:

```
$SIGSOURCE (Minus5V)  
$SIGSOURCE (Plus3V)
```

The \$DESIGNPINSIGSOURCE/\$DESIGNSIGSOURCE mechanism does not provide any way of specifying a field name that is different from the target net name, as can be done with the \$SIGSOURCE command.

To summarize, here are the steps you need to take to use this feature:

Check that a \$DESIGNPINSIGSOURCE/\$DESIGNSIGSOURCE command exists in the netlist script you are using, or add one. The SigSources design attribute field is predefined for this purpose. If you are using a design kit provided with EMTPWorks, check the ReadMe file that came with it for more information.

Place a list of the special signal source fields that you will need in the design attribute field specified above. These items must be separated by commas and each must be a valid attribute field name.

Place the desired pin connections for each of these nets in the corresponding attribute field in each device in your design.

When you produce a Netlist from this design, you should see new nets with the names specified in the design attribute and with the pin connections derived from the values in each device.

I.7. Script Hierarchy Issues

The Export tool has the ability to produce a hierarchical Netlist of a circuit, i.e. a list including the internals of hierarchical blocks. The listing can proceed into nested blocks to any desired depth to provide the required detail. For the following discussions, this facility is referred to as “hierarchical netlisting”. Without hierarchical netlisting enabled, only objects in the current circuit level (i.e. that displayed in the topmost window) will appear in a Netlist.

I.7.1 Types of Hierarchical Netlists

Two types of Netlists can be produced from a hierarchical design:

Flattened Netlist: A flattened Netlist is generated by substituting each hierarchical block symbol with its internal circuit. All information about the hierarchical blocks themselves, nesting levels, etc. is lost. The resulting output appears to have been generated from a flat design. This format is most commonly used for printed circuit board Netlists.

Pure Hierarchical Netlist: A Pure Hierarchical Netlist consists of a Netlist of the master circuit for the design, and then separate Netlists of the internals of each type of hierarchical block used. Each internal circuit can be written into a separate file, or all Netlists may be concatenated in a single file in either top-down or bottom-up order. This format is commonly used for simulators (e.g. SPICE or EMTP) and FPGA tools.

Since a Pure Hierarchical Netlist contains only a single copy of the definition of each internal circuit, it cannot contain any instance data.

I.7.2 The \$HIERARCHY Command

The \$HIERARCHY command in a script file sets the type of Netlist output to be generated.

The hierarchy mode of the Netlist does not have to match that of the design. For example, you can generate a PURE Netlist from a PHYSICAL mode design. The following table indicates which design modes are appropriate for the various Netlist types.

Report Mode	Design Mode	Description
\$CIRCUIT	FLAT, PHYSICAL, PURE	Generate a normal Netlist for the single current circuit only. This is the default mode.
\$TOPCIRCUIT	FLAT, PHYSICAL, PURE	Generate a normal Netlist for the master circuit of the current design only.
\$FLAT	FLAT, PHYSICAL	Generate a flattened Netlist for the entire design.
\$FLATDOWN	FLAT, PHYSICAL	Generate a flattened Netlist for the current circuit and all nested blocks.
\$PURE	FLAT, PHYSICAL, PURE	Generate a pure Netlist for the entire design. In particular, this causes a \$DEVICES listing to list each unique type of device only once.

\$PUREDOWN	FLAT, PHYSICAL, PURE	Generate a pure Netlist for the current circuit and all nested blocks.
------------	-------------------------	--

Note these points on hierarchical format:

- A flattened Netlist will normally make reference to instance attributes in the design. Instance attributes are not normally used in a Flat mode design, so this combination is not recommended.
- PURE netlist mode can be used with a FLAT design but no internal circuits will be listed.
- In PURE Netlist mode, the internal circuit for each block type is listed only once, regardless of how many times it is used in the design. Therefore, instance data is not normally included in a PURE Netlist.
- All hierarchical Netlists are affected by the Report Options settings that restrict entry into sub-circuits. These are described below.

I.7.3 Restricting Reporting of Internal Circuits

The listing of internal circuits of individual blocks can be controlled using the Report Options. The option “Report this device” will cause this device or block to appear in the Netlist as if it has no internal circuit. The option “Report subcircuit” will cause the internal circuit of this device to appear in the Netlist. I.e. in a PURE Netlist, the internal circuit will be defined, in a FLATTENED Netlist, the internal circuit will be substituted and this device will not appear at all.

The “restriction” status is stored in the Restrict device attribute field.

I.7.4 Listing Format for Internal Circuits

For flattened Netlists, the output format is usually the same as would be generated from a flat design, except that hierarchical or instance names (see next section) are sometimes substituted for definition names.

For Pure Hierarchical Netlists, formatting is more complex since the format of the master circuit is usually different from the internal circuit definitions. For this reason, the Report tool treats these two types of circuits as two separate format definitions.

The general outline of a Pure Hierarchical Netlist format is as follows:

```
{
  $PURE mode lists each device type once
  so the next $DEVICES listing will only
  list unique type definitions.
}
$HIERARCHY $PURE
{
  The DEFINECIRCUIT section gives the listing
  format for the internal circuits
}
$DEFINECIRCUIT
{ Listing commands for internal circuit }
$END
```

```

{
    Use FIND command to locate all devices that
    have a non-empty Depth attribute, i.e.
    they have an internal circuit. The Depth
    attribute field contains an integer indicating
    the number of circuit levels below this device
}
$FIND $DEVICES &Depth
{
    Sort by the Depth field to list from the
    bottom up. The $DESCENDING option could
    be used to sort from the top down.
}
$SORT $DEVICES &Depth
{
    The following $DEVICES listing will list
    each unique type of device only once because
    we are in $PURE mode. $INTERNAL causes the
    internal circuit to be output for each device
    in the format defined in $DEFINECIRCUIT.
    Other text or commands could be included in
    the $DEVICES line to output a circuit block
    header or terminator.
}
$DEVICES$INTERNAL
{
    Now switch to TOPCIRCUIT mode and do a
    separate listing for the master circuit
}
$HIERARCHY $TOPCIRCUIT
{ Commands to list master circuit }

```

I.7.5 Depth Ordering in Pure Netlists

The Depth attribute field is used to select device types that have internal circuits and to sort by depth for top-down or bottom-up listings.

The Depth field is not maintained by the Schematic module during normal editing operations. The Depth value is calculated by the Report tool only when a PURE netlist is generated.

The Depth field will be empty for devices with no internal circuit. For devices with an accessible internal circuit, the Depth field will contain an integer indicating the number circuit levels below the device. E.g. if the device's internal circuit contains only bottom-level devices, the value will be 1.

The Depth field can therefore be used with the \$FIND command to perform separate listing operations on devices with or without internal circuits. For example:

```
$FIND $DEVICES &Depth
```

This will select all devices having an internal circuit, whereas:

```
$FIND $DEVICES $NOT &Depth
```

This will select devices having no internal circuit. Similarly, the \$SORT command can be used to determine whether internal circuits are defined from the lowest level up or vice-versa.

```
$SORT $DEVICES &Depth
```

This will cause the lowest level devices to be listed first, i.e. bottom up. This is the preferred order for most formats because it ensures that each type of internal circuit is defined before it is referred to in any other circuit.

I.7.6 Instance vs. Definition vs. Hierarchical Names

Devices and signals have three different types of “names” that are significant in producing Netlists from hierarchical designs. These are summarized in the following table.

Report Keyword	Attribute Field	Description
\$DEVNAME/ \$SIGNALNAME	Name	Normally used in Flat mode designs. Not guaranteed to be unique across a hierarchical design.
\$DEVINSTNAME/ \$SIGNALINSTNAME	InstName	Not normally used in Flat mode designs. InstName is used for package assignment in Physical mode designs. Should be unique across design.
\$DEVHIERNAME/ \$SIGNALHIERNAME	Temporary generated by program	A generated name consisting of the device or signal Name field prefixed by the names of all containing hierarchical blocks.

Note these rules when selecting which type of name to use in a report format:

- \$DEVNAME/\$SIGNALNAME should not be used in a flattened Netlist unless you intend to manually guarantee that it is unique across the entire design and each type of hierarchical block is only used once.
- In Physical mode designs, \$DEVINSTNAME should be used as the device “name” in Netlists intended for PCB-layout use, since it normally contains the package assignment.
- In flattened Netlists, \$SIGNALHIERNAME should be used for the signal “name” since it ensures a unique name that is easily located in the schematic. \$SIGNALINSTNAME can be used but must be assigned manually to each signal.

I.7.7 Power and Ground Connections in Hierarchy

Power and ground symbols (i.e. Signal Connector devices) do not make an immediate logical connection across hierarchy levels. For this reason, power and ground connections in different circuit levels will be considered separate nets in a flattened netlist, unless you do one of the following:

- Specify the name of each power and ground net in a \$SIGSOURCE statement in the script. This instructs the Report tool to look for those named nets and integrate them into a single entity.

-
- Create individual ports for the power and ground connections on each hierarchical block symbol and treat them as regular nets.

1.8. File Input and Output

The Export tool is essentially a text file processing module, so issues of directories, file naming, line terminators, etc. are important considerations in writing scripts. This section discusses some of these issues.

1.8.1 File Names and Paths

Whenever a file name is specified in a script, the issue of which directory the file is going to be found in must be considered. Generally, there are two different kinds of files dealt with by the Export tool:

- Design-specific files: These include the design itself, Netlists or reports generated from it, back-annotation files and possibly "include" files that need to be copied into output of some kind. These files are modified as the design is created and edited by the user.
- System files: These include the scripts used for netlisting and report generation, Prompter setup tables, etc. These files are normally fixed and may be shared by a number of users.

This division of files raises the following issues:

- Most users will tend to keep these two sets of files in different areas. In fact, in network systems or workgroups, the two sets of files might be on quite separate machines.
- When creating scripts and other system files, it is desirable to make them as portable as possible, so that they don't need to be modified to run on a variety of systems. For this reason it is undesirable to include any absolute pathnames in a script.

In order to handle these situations, the Export tool keeps track of two directories at any one time:

- The "current directory" for design-specific files.
- The "root directory" for system files.

These are described in the following sections.

Current Directory

While a script is executing, there is always a "current directory", which is the place where output files will be generated. When a script starts, the current directory will be the one containing the design file associated with the current design. If the current design has not been saved, then the current directory will be the location of the EMTPWorks program.

During the execution of a script, the current directory can be changed by several things:

Explicitly selecting a directory with the \$FOLDER/\$DIRECTORY command.

-
- Creating a new directory with the \$CREATEFOLDER/\$CREATEDIRECTORY command. This sets the current directory to be the new one.
 - Opening a new design. This sets the current directory to be the new design's directory.

Root Directory

The root directory is the starting point for locating system files such as scripts and Prompter tables. For larger installations where the program and common files may be shared over a network, EMTPWorks also permits multiple root directories. The following points should be noted about root directories:

- If not otherwise specified in the Setup or .INI file, the root directory is the one containing the EMTPWorks program.
- The root directory cannot be changed by any script or user commands. It is determined by settings in the Setup or .INI file.
- Names of include files or script names are always given relative to the root directory. The Export tool does not search recursively inside directories. Therefore, any script name must be specified either with an absolute path name (i.e. starting with the disk name), or must be relative to the root directory.
- If there are multiple root directories, the Export tool determines if the "current" directory (described above) is in any of the specified root directories, then it searches first in that one. The purpose of this is to allow users on network systems to have a directory on their local disk which contains their designs and local scripts. If this directory is specified as a root, it will be searched first, before looking in the program location.

When specifying file paths containing backslash characters, remember that the backslash character is also the "escape" character for special symbols in scripts. For this reason, whenever a path is included in text that is interpreted as part of a script, backslashes must be doubled, for example "C:\\Scripts\\Report.rfm".

I.8.2 Types of Output

Output to File

The most common usage of the Export tool is to generate text reports, netlists, etc. For this reason, the default behaviour is to write all text generated by a script to a text output file. If the script does not explicitly create an output file, the user will be prompted to create one with a standard "Save" box as soon as the scripted generates its first character of output. If the script generates absolutely no output, then it will run without this prompt. This behaviour can be changed by using the \$REPORTON/\$REPORTOFF, \$CREATEREPORT and \$CLOSEREPORT commands, described below.

Note that output files can be "nested" to any desired depth. I.e. A script can perform multiple \$CREATEREPORT commands without intervening \$CLOSEREPORT commands. Script output is written to the file created by the most recent \$CREATEREPORT until it is closed by a \$CLOSEREPORT. Output will then go to the next most-recently-created file, etc.

Output to Memory Buffer

The Export tool can also be called by other modules within EMTPWorks to perform various operations. The calling module can specify that script output should be written to a memory buffer for use by that module when the script is completed. In this case, the user will not be prompted to create a file when output is generated. Note that the script can still explicitly create an output file using \$CREATEREPORT. This will be considered a nested output file, as described above.

Transcript Output

The Export tool can support a secondary text output file which is open simultaneously with the main output file. This is referred to as a "transcript file" and is intended as a method of creating an error report or log file that allows the script user to trace errors that occurred during report generation, changes implemented by the script, etc.

Many of the same formatting commands and information sources can be used in writing to the transcript file. Text is written to the transcript file by means of the \$WRITETRANSSCRIPT keyword. The argument to this command is evaluated and the resulting text is written to the transcript file. Note that it is the callers responsibility to insert line terminators where desired using the \$NEWLINE keyword.

I.8.3 Text File Input

The Export tool has the ability to read line-oriented text files and extract arbitrary text data items. This can be used compare a design to an external file or perform simple back annotation tasks. The text file input capability is implemented as a special case of the \$INCLUDE command with the \$EXECUTE option.

I.9. Regular Expressions

The Export tool allows you to use Unix-style regular expressions to check the format of character strings and extract data fields from strings of a known format. Regular expressions make it possible to perform the following kinds of operations:

- Check a data field (e.g. a device or signal name) for correct format. For example, you can easily create a pattern that says "a letter followed 1 to 9 more letters, numbers or underscores". If any items are found that don't match the pattern, you can warn the user of a possible error.
- Convert the format of a data item. For example, suppose you have a schematic that uses bus names of the form "A[0:7]" and you want to create a Netlist for a system that wants to receive a value that looks like "A<7..0>". You can create a regular expression that will extract the parts of the original data item and use them as elements of a new string.
- Extract data from an incoming text file. The Export tool has the ability to read a text file one line at a time and execute a script for each line. You can create a regular expression to match the expected contents of the line and extract data items. These can then be used to set values in circuit objects to perform various kinds of back-annotation.

Regular expressions are invoked using the \$REGEXP command.

I.9.1 The \$REGEXP Command

The \$REGEXP function is the only way to invoke a regular expression. It takes the following form:

```
$REGEXP(regularExpression, string)
```

The *regularExpression* item is a sequence of characters following the syntax described in the next section. The string item is any character string, which can be a literal string, like "VALUE", or a sequence of commands that generate a string value, such as "\$DEVNAME-\$PINNUM".

The \$REGEXP function returns a "false" value (i.e. null string) if the regular expression does not match the string, or a "true" value (i.e. "1") if it does.

The syntax of the function call places a restriction on the use of commas and closing parentheses as literal characters in both the regular expression and the string. If either of these is required as a literal character, they should be preceded by the escape (back-slash) character. This does not apply to closing parentheses used for grouping within the regular expression itself. The parser can detect this usage and will not interpret it as closing the \$REGEXP argument list.

I.9.2 Regular Expression Syntax

A regular expression is simply a sequence of characters that will be compared to another sequence of characters. For example, a string of letters or numbers like "Fred" will match only an identical string of letters or numbers. A small set of punctuation characters have special meaning and are referred to as metacharacters. The regular expression metacharacters are:

\ ^ \$. [] | () * + ?

When any of these items is encountered in an expression, they impart special meaning to one or more of the characters that follow. These meanings are summarized in the following table:

Regular Expression Metacharacters

Format	Meaning
.	Matches any character except newline
[a-z0-9]	Matches any single character of set.
[^a-z0-9]	Matches any single character not in set (in this context ^ means "not").
\d	Matches a digit; same as [0-9]
\D	Matches a non-digit, same as [^0-9]
\w	Matches an alphanumeric (word) character [a-zA-Z0-9_]
\W	Matches a non-word character [^a-zA-Z0-9_]
\s	Matches a whitespace char (space, tab, newline...)
\S	Matches a non-whitespace character
\n	Matches newline
\t	Matches a tab
\nnn	Matches an ASCII character of octal value nnn
\xnn	Matches an ASCII character of hexadecimal value nn
\CX	Matches an ASCII control character

<code>\metachar</code>	Matches the character itself, i.e. overrides normal meaning of special characters. E.g. "[" introduces a set, but "\[" matches "["
<code>(abc)</code>	Provides grouping and remembers the match for later backreferences and match variables. See below.
<code>\n</code>	n is a decimal digit from 1 to 9. Matches whatever the nth set of parentheses matched
<code>x?</code>	Matches 0 or 1 x's, where x is any of above
<code>x*</code>	Matches 0 or more x's
<code>x+</code>	Matches 1 or more x's
<code>x{m,n}</code>	Matches at least m x's but no more than n
<code>abc</code>	Matches all a, b, and c in order
<code>fee fie foe</code>	Matches any of fee, fie, or foe

The increasing precedence of operators is alternation, concatenation and unary (*, + or ?). The repetition characters (*, + and ?) all match as many characters as possible before proceeding to the right. For example, in the expression:

```
.*.*
```

The first .* will always match the entire string and the second .* will match the empty string. This can be particularly significant when using match variables as it will affect which portion of the string gets assigned to which match variable.

I.9.3 Match Variables

A powerful feature of regular expressions is the ability to “remember” what string of characters a specific part of the expression matched. Whenever you enclose part of a regular expression in parentheses, you have implicitly created a match variable. Because the syntax of regular expressions only allows one decimal character for a match variable name, there are exactly 9 of them, named “1”, “2”, “3”, etc. As a regular expression matching operation proceeds, the string of characters in the subject string that matched the first set of parentheses in the expression is assigned to 1. Whatever matches the second set of parentheses is assigned to 2, etc. If parentheses are nested, the inner set will be the higher number.

I.9.4 Back-references Within an Expression

You can refer back to a match variable within an expression using the format “\n”. This, in effect, has the meaning “match the same stuff that was matched by the nth set of parentheses again”. This can be used to look for repeating patterns. For a simple example:

```
(.+) \1
```

It matches any string that has any sequence of characters repeated twice. For example, any of the following would match:

```
XX 123...123... _1__1_ FredAndMaryFredAndMary
```

I.9.5 Match Variable References Outside an Expression

After the \$REGEXP command has completed execution successfully (i.e. the pattern matched), the values of the match variables are available for use in the script. This is done by referring to the variables "1" through "9" as &1 through &9.

For example, the following can be used to extract the numeric part of a name:

```
The number is $IF($REGEXP(.*(\d+), $DEVNAME))&1$END
```

In this case, if the name has no numeric part, nothing will be output. An \$ELSE clause on the \$IF could be used to perform some other operation.

I.9.6 Differences from Unix Regular Expressions

For users familiar with Unix regular expressions, there are only a couple of minor differences, imposed by the structure of a command file:

The ^ and \$ operators that match the beginning and end of a line are not implemented. In this implementation, the regular expression must always match the whole subject string, so they are superfluous.

A comma character "," must be escaped when used in a regular expression, because it will otherwise terminate the argument string to the \$REGEXP command. I.e. if you need to match a comma, you must put "\", in the regular expression.

I.9.7 Regular Expression Examples

This section offers a number of examples of using regular expressions for error checking and for extracting or reformatting text data.

Checking for Numeric Data

Regular expression alone:

```
|.*\D.*
```

Simple script usage:

```
$IF($REGEXP(|.*\D.*, $PINNUM))$SETVAR(_Error, 1)$END
```

This expression matches any text string that is either empty (that is, zero length) or contains any non-numeric character. This could be used, for example, to check for invalid pin number data during netlist generation. This example looks a bit strange because it starts with an "|" alternation operator that is supposed to work on two operands. In this case, the left hand operand is nothing, or the empty string. The right hand operand of the "|" is ".*\D.*", in other words "any string of zero or more characters, followed by a single, non-decimal character, followed by any string of zero or more characters".

Checking for Invalid Characters in Names

Regular expression alone:

```
[a-zA-Z_]+
```

Simple script usage:

```
$FIND $DEVICES $NOT($REGEXP([a-zA-Z0-9_]+, $DEVNAME))
```

This example makes use of the "[]" operator to list the allowable characters in a device name, in this case letters, numbers and the underscore character.

Extracting Data Fields

Regular expression alone:

```
([A-Za-z0-9_$\-]+) .*
```

Simple script usage:

```
$IF($REGEXP(([A-Za-z0-9_$\-]+) .*, $BUSNAME)) &1$END
```

This example is making use of extra parentheses to assign parts of the matched text to match variables. The "[A-Za-z0-9_\$\-]+" portion of the expression matches any string of one or more of the characters listed in between "[" and "]". The parentheses around this part of the expression cause the result of this portion of the match to be assigned to match variable number 1. The remaining portion of the expression ".*" will match all remaining characters in the target string.

The script example shown was created to take bus names like "DATA[0,15]" and extract only the name from the front. In this case, once the expression has matched, the script variable &1 will have the value of the portion of the text that matched the portion of the expression in parentheses.

I.10. Script Examples

I.10.1 EMTP Netlist.rfm

The file “EMTP Netlist.rfm” contains the script used by EMTPWorks to generate the EMTP data Netlist. The following lines paragraphs provide step by step explanations on this script:

```
{  
    EMTP Netlist Generator  
    C. Dewhurst - Feb 18, 2003  
}
```

Comments are enclosed between left and right braces, “{ and }”.

```
$CREATEREPORT($DESIGNNAME.net)
```

Creation of the report output file. The output file will be named \$DESIGNNAME.net. The keyword \$DESIGNNAME returns the current design’s name.

```
$BUSNAMEON()
```

This indicates that the name of the bus (bundle) will be added for any reference of a signal name in a bundle.

```
$LINEWIDTH(100)
```

This sets the maximal number of characters to output in one line of the file, i.e. in this case 100.

```
$CONTEND(,)
```

This specifies a character to be added at the end of a line that has to be wrapped.

```
$CONTSTART()
```

This specifies a character to be added at the beginning of a wrapped line.

```
$SETVAR(_Errors, ){ This will become non-blank if any errors found }
```

This sets a new variable named _Error. It is initialized to null.

```
$DEFINEATTR(PhaseTemp/DM)  
{ Define a temporary field for use calculating device phases }
```

This sets a new attribute. The letter D means that this attribute is for a device and the letter M means that is a temporary field. For more information on others specifications, see \$DEFINEATTR.

```
! Generated by EMTPWorks version $DWVERSION $DATE $TIME  
!  
Circuit.Diagram=$FILENAME;  
Circuit.Date=$DATE($d/$m/$Y) $TIMEMODIFIED($h:$n:$s);
```

If the text is not a command, it is put as raw text in the report. \$DWVERSION returns a string with the current EMTPWorks version. \$DATE and \$TIME return respectively the current date and time. \$TIMECREATED/\$TIMEMODIFIED return the creation and modification times respectively.

```
$IF(&ParamsEMTP)  
!  
&ParamsEMTP
```

\$END

The “&” symbol is used to access variables and attributes. The following script means that if the attribute ParamsEMTP is non-null, “!” character followed by the ParamsEMTP design attribute is written in the report.

The next lines are used to define a subroutine that can be called from anywhere in the code.

```
{
This block defines the output for each device line. 3-phase items
(Phase attribute is "3") are replicated for each phase.
}
$DEFINEBLOCK (DevNet)
```

A block named DevNet is defined. This block is like a function, it contains lines that will be executed later in the script for listing devices. This block ends when the command \$END is reached. The block allows listing the device signal names using the pin format. Each pin is connected to a signal.

The block has several sections for each type of device Netlist. Currently there are 2 types of signal formats to be treated: general (1-phase) and 3-phase. Since the GND signal is treated as a blank character in the Netlist, it must be tested separately. A general signal may be used as a phase signal in which case it must be appended with the phase character ‘a’, ‘b’ or ‘c’. This character is available for 1-phase signals through the keyword \$EMTPPHASE. In all other non 3-phase cases, \$EMTPPHASE is empty.

```
$IF (&Depth) { Format for sub-circuit devices }
```

This condition is true if the device’s attribute Depth is non-null. This attribute is non-null if the device is in a sub-circuit. The following lines define the pin format for subcircuits.

```
$DEVPINFORMAT
(
$IF ($GT ($PINSEQ, 0) ) $CONTEND (, ) $END
$IF ($EQ ($EMTPPHASE, 3) )
$IF ($EQ ($SIGNAME, GND) ) , , $ELSE $SIGNAME\`a`, $SIGNAME\`b`, $SIGNAME\`c` $END
$ELSE
$IF ($NE ($SIGNAME, GND) ) $SIGNAME$EMTPPHASE $END
$END
)
```

The pin names are comma separated. A comma is added at the end of the wrapped line if the device has pins and only after listing at least one pin.

The following lines provide the complete subcircuit device format.

```
$IF (&Visual.Dev) | $ELSE @ $END &PartTemp;
$DEVNAME;
$NUMEMTPPINS;
$CONTEND () $PINS $CONTEND (, ) ,
$IF (&ParamsA) $NEWLINE &ParamsA $END
$IF (&ModelData) $NEWLINE &ModelData $END
```

A different first character is used to distinguish between visual and standard subcircuits.

\$DEVNAME returns the device’s name.

\$NUMEMTPPINS returns the real number of pins (a 3-phase pin is counted 3 times).

\$NEWLINE inserts a new line. The attributes &ParamsA and &ModelData are used for saving device data and listed only if non-empty. These will be available for masked devices.

A typical example is given by:

```
@3p_test_a75f5bc8;DEV2;8;s61a,s61b,s61c,s62a,s62b,s62c,s59b,s60,
```

The device DEV2 has 8 pins. There are two 3-phase pins: s61 and s62. The signal s59b is connected to the phase b of a 3-phase signal. The signal s60 is a general signal.

The following lines are for non-subcircuit devices.

```
$ELSE
```

```

$IF($EQ(&PhaseTemp, 3))
    { Format for 3-phase, non-sub-circuit devices }
$DEVPINFORMAT (
$IF($GT($PINSEQ,0))$CONTEND(,) $END
$IF($NE($SIGNAME,GND))
    $SIGNAME
    $IF($EQ($EMTPPHASE,3))
        ~
    $ELSE
        $EMTPPHASE
    $END
$END)

```

In the case of 3-phase devices it is needed to replicate the device for each phase. The format listing is first saved into a temporary variable `_TEMP` and then used in the replication process.

```

$SETVAR(_TEMP,\_&Part\;$DEVNAME~\;$NUMEMTPPINS;$Numpins;$CONTEND()$PINS$CONTEND(,)\,)
$REPLICATE(&_TEMP,~, , a)
$IF(&ParamsA)$NEWLINE&ParamsA$END
$IF(&ParamsExtra),&ParamsExtra$END
$NEWLINE
$REPLICATE(&_TEMP,~, , b)
$IF(&ParamsB)$NEWLINE&ParamsB$END
$NEWLINE
$REPLICATE(&_TEMP,~, , c)
$IF(&ParamsC)$NEWLINE&ParamsC$END
$IF(&ModelData)$NEWLINE&ModelData$END

```

`$REPLICATE` replaces the `~` character by letters a, b, or c in the variable `_TEMP`. A typical example is given by this 3-phase RLC device:

```

_RLC;RLC1a;2;2;s2a,s6a,
1,0,0,0,0,
_RLC;RLC1b;2;2;s2b,s6b,
1,0,0,0,0,
_RLC;RLC1c;2;2;s2c,s6c,
1,0,0,0,0,

```

The device is replicated 3 times and its name and signals are appended with the applicable phase character. If a given device has 3-phase and 1-phase signals (pins), then its 1-phase signals are listed (repeated) on all phase lines without appending the phase character. It is noticed that only power pins can become 3-phase.

If a device has no 3-phase signals, then it is listed using the code below:

```

$ELSE
    { Format for single-phase, non-sub-circuit devices }
$DEVPINFORMAT (
$IF($GT($PINSEQ,0))$CONTEND(,) $END
$IF($NE($SIGNAME,GND))$SIGNAME$EMTPPHASE$END)
$IF(&Depth)$IF(&Visual.Dev)|$ELSE@$END$END\_&Part;
$DEVNAME;
$NUMEMTPPINS;
$Numpins;
$CONTEND()$PINS$CONTEND(,)
$IF(&ParamsA), $NEWLINE&ParamsA$END
$IF(&ModelData)$NEWLINE&ModelData$END
$END
$END
$END

```

The following lines are used to create a block for the basic circuit format.

```
{
    This block defines the basic circuit format that is used
    by the top-level circuit and all subcircuit blocks
}
$DEFINEBLOCK(Circuit)
{
    Run through devices and figure out which ones are 3-phase
    (i.e. have at least one 3-phase pin)
}
$REPORTOFF
$DEVPINFORMAT $IF($EQ($EMTPPHASE, 3))3$END
$FIND $DEVICES          { Scan all devices }
$ITEMSEPARATOR()
$DEVICES $PINS $SETATTR(PhaseTemp, $IF($NONBLANK($PINS)) 3$END)
$REPORTON
$BUSNAMEON()
{
    Set formats for regular device listing
}
$ITEMSEPARATOR(,)
{
    Print the main device list for this circuit level
}
$FIND $DEVICES $NOT(&Exclude)
$SORT $DEVICES &Depth &Part &PartTemp $DEVNAME
$FIND $SIGNALS
$PROGRESS(Writing netlist) $PERCENTON
$DEVICES $INLINE(DevNet)
$END
```

The command \$REPORTON/\$REPORTOFF when OFF means that all the following lines will not be written into the report until the command \$REPORTON appears.

The first script lines are used to find the 3-phase devices.

Before listing the devices it is needed to exclude those that are excluded in the design: the &Exclude attribute is used in the design to indicate an excluded device.

The \$SORT command sorts the devices. The subcircuits are listed first, and then the devices are sorted by the Part or PartTemp attributes or by device name.

After finding all signals, the devices are listed using the DevNet block. The command \$INLINE means that the script in the block will be executed like if it was appearing on one line.

The command \$DEFINECIRCUIT indicates the start of an internal circuit definition. The end of this definition is set by the \$END command. This block is referred by the \$INTERNAL command. This script will be used to output the internal circuit of all devices that contain the \$INTERNAL command. The previously defined Circuit block is reused through the \$INCLUDE command.

```
{
    This defines the format for a subcircuit
}
$DEFINECIRCUIT
$LINEWIDTH(100)
$CONTEND(,)
$CONTSTART()
$INCLUDE $BLOCK(Circuit)
$END
```

At this stage everything is ready to start the actual Netlist printing for all devices. The first step is to list the simulation option devices. These are notified by the &Function attribute value OPTION.

```
{
    Display general option devices, distinguished by OPTION
    in the Part attribute field

    NOTE: Only items in the top-level circuit are listed.
    Options devices in subcircuits will be ignored.
}
$HIERARCHY $TOPCIRCUIT
$FIND $DEVICES $REGEXP(OPTION.*, &Function)
$IF($GT($DEVCOUNT, 0))
!
! The following lines are derived from option symbols
! on the schematic
!
$DEVICES$IF(&ParamsA)&ParamsA$IF(&ModelData)$NEWLINE&ModelData$END$ELSE&ModelData$END
$END
```

\$HIERARCHY \$TOPCIRCUIT command means that only OPTION devices from the topmost circuit in the current design are accepted.

Next script lines are for the standard devices starting with subcircuit definitions.

```
{
    This is where the actual output generation starts.

    Put out the subcircuit definitions
    $PURE script mode lets us list each internal circuit once
    While we're at it, make sure a value has been specified in the Part attribute.
    If not, use the type name. This isn't an issue for library parts, but some
    users (your humble authour, for example) may produce a subcircuit block
    and forget to put a value in the Part field.
}
$HIERARCHY $PURE
```

See **\$HIERARCHY** command reference for more details on the **\$PURE** option.

```
$FIND $DEVICES &Depth
$IF($GT($DEVCOUNT, 0))
!
! Subcircuit definitions
!
$REPORTOFF
$DEVICES$SETATTR(PartTemp,
$IF($NOT(&Part)) $TYPENAME$ELSE&Part$END\_$_HEX($CHECKSUM))
```

The part name Part is completed using a hexadecimal code to create PartTemp. See the **\$CHECKSUM** command reference for more information.

The subcircuit devices are sorted using the &Depth and &PartTemp attributes. The command **\$INTERNAL** calls the **\$DEFINECIRCUIT** script.

```
$REPORTON
$SORT $DEVICES &Depth &PartTemp
$COMBDEVSON
$DEVPINFORMAT
$IF($EQ($CHILDEMTPPHASE, 3))
    $CHILDSIGNAMEa, $CHILDSIGNAMEb, $CHILDSIGNAMEc
$ELSE
    $CHILDSIGNAME$CHILDEMTPPHASE
$END
$ITEMSEPARATOR(, )
```



```
$DEVICES<&PartTemp;$NUMEMTPPINS;$PINS$SINGLE,$INTERNAL>
!
!
$END
```

An example of subcircuit definition is given by:

```
<3p_test_a75f5bc8;8;s52a,s52b,s52c,s57a,s57b,s57c,s54,s55,
_RLC;R2a;2;2;s52a,s57a,
1,,,,,
_RLC;R2b;2;2;s52b,s57b,
1,,,,,
_RLC;R2c;2;2;s52c,s57c,
1,,,,,
_RLC;R3;2;2;s54,s55,
1,,,,,
>
```

All interface pins are appearing following the subcircuit identification and number of pins.

At this stage the hierarchy is set to TOPCIRCUIT to start listing devices that are not located in subcircuits.

```
{
    Here is where the actual top-level report starts
}
$HIERARCHY $TOPCIRCUIT
$COMBDEVSOFF
{
    Do the main top-level listing
}
$INCLUDE $BLOCK(Circuit)
```

The script from the block Circuit is executed to list the devices of the top-level circuit.

At this stage the entire Netlist has been completed. The following lines are used to optionally open the Netlist in the EMTPWorks text editor. This option is selected by setting the first argument of the complete script to OPEN.

```
{
    If global params indicate we want an open, do that now
}
$IF($EQ(OPEN, &ARG1))
$CLOSEREPORT
$TEXTOPEN($DESIGNNAME.net)
$END
```

It is also allowed to immediately start the EMTP program after creating its Netlist.

```
{
    If global params indicate we want EMTP started, do that now
}
$IF($EQ(EMTP, &ARG1))
$CLOSEREPORT
$SYSTEMOPEN(empt/emptopt.exe, "emptprv.ini;emptprvstate.ini;$DESIGNPATH$DESIGNNAME.net;1;")
$END
```

The above script "EMTP Netlist.rfm" is called from JavaScript codes attached to EMTP menu items. You can identify the scripts by looking into the EMTP.INI file section on menus.

II. Part II – Script Keyword Reference

This part of the manual lists all the command keywords defined in the script language, in alphabetical order.

Many of the keywords described here were added to the language to support specific Netlist formats or customer requirements. We do not guarantee that all keywords and commands have been tested in every possible combination. When using these commands to create critical Netlists or reports, please verify your results carefully before relying on them!

For each keyword, the following information is provided:

- **Keyword**—the keyword as it should appear in the script. In some cases, two or more related keywords are grouped together.
- **Status**—this indicates whether any changes have been made since a previous version.
- **Synopsis**—A short summary of the usage of the keyword, i.e. what arguments it takes. Note that the following meta-characters are used to indicate optional or alternative constructions in the synopsis:
 - `[]` surround an optional item, e.g. `$DATE[(format)]`.
 - `|` indicates an alternative, for example `$SORT $DEVICES|$SIGNALS`.
 - *italics* indicate an item that should be replaced by text performing the given function, e.g. `msgText` will be replaced by the actual argument text.
 - **bold type** indicates literal text that should be included exactly as given.
- **Returns**—a description of the "return value" of the keyword, i.e. what value will get substituted for it in the output file. N.A. (not applicable) indicates that this keyword has no value.
- **Type**—this will be either "Data", "Action", or "Definition". This distinction determines how the keyword is handled if it appears by itself on a line. If a line of the script contains only Definition or Action keywords, then no line terminator is written out to the file. If the line contains one or more Data keywords, then the line is considered to be output data and a line terminator will be written out after all data generated by the keywords.
- **Where**—this indicates where the keyword can be used. The following locations are specified:

Top Level	The keyword must be on a line by itself, starting in column 1. This keyword can operate successfully without any current design.
Top Level - Design	The keyword must be on a line by itself, starting in column 1, and there must be a current design available in order for the keyword to obtain the data necessary to calculate its value or perform its function.
Script	The keyword can be in any executed part of the script, i.e. not in a value mapping table or other areas that are interpreted as raw data. The keyword does not need to refer to any object to perform its function.
Script – Design	As for Script, except that there must be a current design.
Script – Device	As for Script, except that there must be a current device at the position the keyword is at in the script.

Script – Signal	As for Script, except that there must be a current signal at the position the keyword is at in the script.
Script – Pin	As for Script, except that there must be a current pin at the position the keyword is at in the script.

Keyword	1 \$ALERT1/\$ALERT2
Status	
Synopsis	\$ALERT1(<i>message</i>)
Returns	Boolean
Type	Data
Where	Script
Description	Displays an alert box to the user with the given message text. \$ALERT1 displays only one button, OK. \$ALERT2 displays two buttons, OK and Cancel. Returns TRUE for OK, FALSE for Cancel.
Example	\$ALERT1(\$TIME is too late for this kind of thing. Go home!)

If \$ALERTx is used on a line by itself, it will insert the value "1" into the output file if OK is pressed, which may not be desired. You can prevent by enclosing the whole thing in a \$NULL command, as in:

```
$NULL($ALERT1(Warning: Some value changes were made.))
```

➤ See also: \$PROMPT1/\$PROMPT2

Keyword	2 \$ALIGNCOLSON/\$ALIGNCOLSOFF
Status	
Synopsis	\$ALIGNCOLSON \$ALIGNCOLSOFF
Returns	N.A.
Type	Definition
Where	Top Level
Description	Turns on/off automatic column alignment.
Example	\$ALIGNCOLSON

When ON, it causes extra blanks to be inserted between netlist or component list entries. If \$ALIGNCOLS is OFF, then the item separator can be a tab if \$TABFIELDS is ON, or a single blank otherwise. Default is OFF.

The column spacing that is used by the \$ALIGNCOLSON option is set by the \$SPACE command.

You may have to insert a number of blanks at the front of continuation lines using \$CONTSTART to force correct alignment of the first item in each continuation line.
See also: \$SPACE.

Keyword	3 \$AND
Status	
Synopsis	\$AND(string1,string2)
Returns	Boolean
Type	Data
Where	Script
Description	Performs a logical AND operation on its two arguments. Any non-null string in an argument is considered TRUE.
Example	\$IF(\$AND(&X1, &X2))&X1&X2\$ELSE&X1&X2\$END

Keyword	4 \$ASSIGNINSTNAMES
Status	
Synopsis	\$ASSIGNINSTNAMES \$DEVICES \$SIGNALS <i>format</i>
Returns	N. A.
Type	Action
Where	Top level - Design
Description	Assigns default names to the InstName field of any device or signal not having one.
Example	\$ASSIGNINSTNAMES \$DEVICES

The \$ASSIGNINSTNAMES command has exactly the same format and operation and the \$ASSIGNNAMES command except that it operates on the InstName field instead of the Name field. It is intended for use when producing flattened netlists from designs created in the Physical Hierarchy mode. The Physical Hierarchy mode allows the InstName field to take on a different value for each physical device represented by a design, and so can be made unique throughout a

hierarchical design. By contrast, the Name field is only guaranteed to be unique within a single circuit block and will be the same inside all instances of the same sub-circuit.

For a complete format summary, see: \$ASSIGNNAMES.

Keyword	5 \$ASSIGNNAMES
Status	
Synopsis	\$ASSIGNNAMES \$DEVICES \$SIGNALS <i>format</i>
Returns	N. A.
Type	Action
Where	Top level - Design
Description	Assigns default names to the Name field of any device or signal not having one.
Example	\$ASSIGNNAMES \$DEVICES

The \$ASSIGNNAMES command is used to apply names to unnamed devices or signals in the circuit before a listing is generated. If this is not done, any object which has not been named (either manually or by the auto-naming or packaging features) will appear as "unnamed" in a net or component list.

The names assigned by \$ASSIGNNAMES will be invisible but permanently associated with the object. They can be made visible either using the Attributes command associated with the object or the Browser tool.

Note that \$ASSIGNNAMES can be used in conjunction with the \$FIND command to assign names to selected subsets of devices, such as all resistors, etc. See the Examples section for more information.

The "object type" specification (either "\$DEVICES" or "\$SIGNALS") is mandatory, all other items are optional.

Note that the same format and options apply to the \$ASSIGNINSTNAMES command.

Note the following examples:

```
$ASSIGNNAMES $DEVICES $TYPENAME $FORMAT(format) &prefixField
$ASSIGNNAMES $SIGNALS &prefixField
```

The \$TYPENAME, \$FORMAT and &prefixField items are all optional and specify how a name is to be generated. If none of these items appears on the line, then the default format will be used. For devices, this is the contents of the prefix field plus an integer. For signals, it is the specified default signal prefix plus an integer.

\$TYPENAME indicates that the device's type name (i.e. the name that appears in the Parts palette) is to be used as a prefix. If a \$FORMAT appears as well, or if the default format is used, the type name is prefixed to the name that would be generated by the \$FORMAT alone.

&prefixField specifies an attribute field to be used as a source for a prefix. If all three format specifications are given, all three prefixes will be concatenated.

In normal circuits, we do not recommend specifying the following \$FORMAT option in \$ASSIGNNAMES or \$ASSIGNINSTNAMES when assigning to signals. If you specify a format that is anything other than the default, the Schematic tool will assume these are fixed, user-assigned names and will not reassign them during editing operations. This can result in errors if a section of schematic is copied, creating duplicate names.

\$FORMAT specifies a format string consisting of an alphabetic part followed by a numeric part. The alphabetic part will be used as the prefix for every name generated. The numeric part is used simply to specify the number of digits to be used. For example, the default format "D00000" indicates that all names are prefixed with "D" and the numeric part will have a fixed length of 5 digits. Extra digits are added to the numeric part if needed, so "U0" will generate U123 for the 123rd device.

➤ See also: \$UNNAMEDDEVS \$UNNAMEDSIGS \$ASSIGNINSTNAMES

Keyword	6 \$AUTONUMBER
Status	
Synopsis	\$AUTONUMBER(number of pins)
Returns	N. A.
Type	Definition
Where	Top-Level
Description	Specifies a number of pins less than or equal to which a device will have pin numbers automatically assigned in the report output for any reference to \$PINNUM.
Example	<p>\$AUTONUMBER(3)</p> <p>Will automatically provide a pin number whenever a \$PINNUM command appears for an unnumbered device pin on a device which has 3 or less pins.</p>

Any device with less than or equal to the number of pins specified will have pin numbers automatically substituted if the \$PINNUM keyword is used and no pin numbers are present in the circuit file. This option is intended to provide pin numbers for discrete components in a circuit, since they do not normally have pin numbers on a diagram. The default number is zero. Note that this option does not make any change to the circuit data itself, but is simply a substitution that is made during report generation.

Keyword	7 \$BLANKREPLACE
Status	
Synopsis	\$BLANKREPLACE(<i>string</i>)
Returns	N. A.
Type	Definition
Where	Top-Level
Description	Specifies a string of characters that will be substituted for a blank in any device or signal name.
Example	<p>\$BLANKREPLACE(_)</p> <p>Will replace each blank in a name with an underscore character.</p>

The \$BLANKREPLACE item specifies a string of characters that will be substituted for a blank in any device or signal name. This is done to accommodate systems which cannot accept blanks in names.

➤ See also: \$CHARMAP.

Keyword	8 \$BREAK
Status	
Synopsis	<code>\$BREAK(<i>blockName</i>) \$DEVICES \$SIGNALS [\$FIRSTON/OFF] <i>format</i></code>
Returns	N. A.
Type	Definition
Where	Top-Level
Description	Sets up a value break condition for a subsequent \$DEVICES or \$SIGNALS listing.
Example	<code>\$BREAK(PartsHdr) \$DEVICES &Value</code> Will insert the contents of block "PartsHdr" at the front of the next \$DEVICES listing and again every time &Value changes.

The \$BREAK command is used to create listings such as parts lists in which a title block or page break is inserted between groups of lines that are related by some value. For example, you could create a listing in which capacitors are grouped together, resistors are grouped together, etc.

Following is a typical example of usage of this feature:

```
$DEFINEBLOCK(PartsHdr)

-- Here are the parts of type &Function

$END
$SORT $DEVICES &Function $DEVNAME
$BREAK(PartsHdr) $DEVICES &Function
$DEVICES$DEVNAME &Value
```

Before each line is generated, the Scripter evaluates the format string, in this case "&Function" and compares the current value to the value in the last line. If the value is different, the contents of the block "PartsHdr" are evaluated and written to the output.

Notes on the \$BREAK command:

- Before performing the listing, you must have specified a \$SORT on the same value that is used for the break, or the listing will not make much sense. This is not checked by the program. If you have not sorted, the program will simply insert a header whenever two adjacent lines have different values of the format string.
- While the header block is being interpreted, the "current object" is the device or signal that is about to be listed. Thus, if you make reference to an attribute or command that refers to an object, it will use the values that would be in effect for the next line.
- You can control whether a header is inserted before the first line using the \$FIRSTON/ \$FIRSTOFF keywords. See below.

A break remains in effect for all subsequent listings once it is established. To turn it off again, you can specify a null format string, as in:

```
$BREAK(PartsHdr) $DEVICES
```

Optional Header Insertion Before First Break

The optional keywords \$FIRSTON/\$FIRSTOFF can be used to control whether a break header is inserted before the start of the listing. If the \$FIRSTON keyword appears immediately after \$DEVICES or \$SIGNALS, then the break header will be inserted immediately before the first device or signal line. If \$FIRSTOFF appears, no header will be inserted until the first value change. The default is \$FIRSTON.

➤ See also: \$HEADER.

Keyword	9 \$BUSCLOSE
Status	
Synopsis	<code>\$BUSCLOSE(string)</code>
Returns	String
Type	Data
Where	Script – signal
Description	Returns the given string if the current object is in a named bus.
Examples	<code>\$SIGNALS\$BUSNAME(I)\$SIGNAME\$BUSCLOSE(I)</code> This will generate a list of the signals in the circuit. If a signal is in a bus, then it will appear as "busName[sigName]", otherwise it will just appear as "sigName".

➤ See also \$BUSNAME.

Keyword	10 \$BUSNAME
Status	
Synopsis	<code>\$BUSNAME[(suffixStr)]</code>
Returns	Text
Type	Data
Where	Script – signal
Description	Returns the name of the bus associated with the current object. An optional suffix string allows this keyword to be used with \$SIGNAME to provide a compound name when the signal is in a bus.
Examples	<code>\$SIGNALS\$BUSNAME(_)\$SIGNAME</code> This will generate a list of the signals in the circuit. If a signal is in a bus, then it will appear as busName_sigName, otherwise it will just appear as sigName.

The naming rules in EMTPWorks allow signals with the same name to exist in multiple different busses without being connected. This means that any report output referring to \$SIGNAME risks being ambiguous if busses have been used in the design. For this reason, it is best to qualify signal names with the name of any enclosing bus. \$BUSNAME allows this by inserting the name of any enclosing bus and an optional separator character in the output.

Because the given separator string is only inserted if the bus name is non-null, \$BUSNAME can always be inserted in front of the \$SIGNAME command without interfering with non-bussed signals. \$BUSNAME can also be used in conjunction with \$BUSCLOSE to give names that are enclosed in parentheses or similar separators. For example:

```
$SIGNALS$BUSNAME (<) $SIGNAME$BUSCLOSE (>)
```

will generate signal names of the form DATABUS<D0>. To take this example one step further, note how we can use a regular expression to create a vector notation for signal names with a numeric component (note that this must all appear on one line in the script):

```
$IF($AND($BUSNAME, $REGEXP(\D*(\d+), $SIGNAME))) $BUS-  
NAME<&1>$ELSE$SIGNAME$END
```

For a signal called A13 in a bus called ADDR, this will generate ADDR<13>. For non-bussed signals, it will simply output the signal name.

➤ See also: \$BUSNAMEON/\$BUSNAMEOFF.

Keyword	11 \$BUSNAMEON/\$BUSNAMEOFF
Status	
Synopsis	\$BUSNAMEON[(prefixStr[,suffixStr])] \$BUSNAMEOFF
Returns	N. A.
Type	Definition
Where	Top-Level
Description	Causes the name of the enclosing bus to be automatically inserted whenever \$SIGNAME is used referring to a signal in a bus. If the bus exists, the prefixStr text is inserted immediately after the bus name and the suffixStr text is inserted immediately after the signal name. This setting remains in effect for all listings until a \$BUSNAMEOFF is encountered.
Examples	<p>\$BUSNAMEON(\[,])</p> <p>Specifies that all references to the \$SIGNAME of a bussed signal will now be listed as busName[sigName]. Note: The escape (backslash) character used in front of each of the arguments is not strictly necessary in this case but is used to clarify that these are intended as literal text.</p>

➤ See also : \$BUSNAME \$BUSCLOSE.

Keyword	12 \$BUSPINCLOSE
Status	
Synopsis	\$BUSPINCLOSE(<i>string</i>)
Returns	String
Type	Data
Where	Script – pin
Description	Returns the given string if the current pin is an internal pin inside a bus pin.
Examples	See \$BUSPINNAME

➤ See also \$BUSPINNAME.

Keyword	13 \$BUSPINNAME
Status	
Synopsis	\$BUSPINNAME[(suffixStr)]
Returns	Text
Type	Data
Where	Script – pin
Description	Returns the name of the bus pin associated with the current bus internal pin, or null if the current pin is not a bus internal pin. An optional suffix string allows this keyword to be used with \$PINNAME to provide a compound name when the pin is in a bus.
Examples	<pre>\$DEVPINFORMAT\$BUSPINNAME(())\$PINNAME\$BUSPINCLOSE()</pre> <p>This sets the format for a pin listing so that if a pin is in a bus pin it will appear as "busPinName[pinName]", otherwise it will appear as "pinName".</p>

The naming rules in EMTPWorks allow pins with the same name to exist in multiple different bus pins without being connected. This means that any report output referring to \$PINNAME risks being ambiguous if bus pins have been used in the design. For this reason, it is best to qualify pin names with the name of any enclosing bus pin. \$BUSPINNAME allows this by inserting the name of any enclosing bus pin and an optional separator character in the output.

Because the given separator string is only inserted if the bus pin name is non-null, \$BUSPINNAME can always be inserted in front of the \$PINNAME command without interfering with non-bussed pins. \$BUSPINNAME can also be used in conjunction with \$BUSPINCLOSE to give names that are enclosed in parentheses or similar separators.

➤ See also: \$BUSNAME \$BUSPINCLOSE.

Keyword	14 \$CALLTOOL
Status	
Synopsis	\$CALLTOOL(toolName, [argumentString])
Returns	Boolean
Type	Data
Where	Script (Note: Whether or not an object is required depends on the tool)
Description	Transfers control to another EMTPWorks tool (MEDA module) and passes some argument information. Returns TRUE if the tool returns success, FALSE otherwise.
Examples	<pre>\$CALLTOOL(Prompter, &FileName)</pre> <p>Calls the Prompter tool and passes the contents of the FileName attribute or variable as an argument.</p>

This command was implemented specifically to allow the Scripter to work with the Prompter, but can also be used to invoke any other tool. The effect of this command is the same as if the tool was invoked by selecting its name in the Tools menu. Note that the "toolName" argument that is the string known as the tool's "alias" that is hard-coded when the tool is created. In most cases these are the same, but they may be different if any file names have been changed or if EMTPWorks has been localized.

Keyword	15 \$CHANGECOUNT
Status	
Synopsis	\$CHANGECOUNT
Returns	Decimal integer.
Type	Data
Where	Script – Design
Description	The number of changes (i.e. editing operations) performed on the circuit since it was created, represented as a decimal integer. This number can be used to check for matching versions of a circuit file.
Example	<pre> \$IF(\$NE(\$CHANGECOUNT, &OldChange)) \$SETATTR(&OldChange, \$CHANGECOUNT) \$NULL(\$ALERT1(The design has changed!)) \$END </pre> <p>This will display an alert if the design has been edited since the last check.</p>

Keyword	16 \$CHARMAP
Status	
Synopsis	\$CHARMAP(blockName, string)
Returns	Text
Type	Data
Where	Script
Description	Used to map the characters making up a string using a predefined table.
Example	<pre> \$DEFINEBLOCK(CharTable) . _DOT_ / _SLASH_ & _AMPERSAND_ \$END \$DEVICES\$CHARMAP(CharTable, \$DEVNAME) </pre> <p>This will output device names, mapping characters according to the given table. For example, a name "U1.1" will be output as "U1_DOT_1".</p> <p>Note that the white space shown in this example is representing a single tab character. Blanks do not count as separators and may be included in the values.</p>

This command allows you to map characters in strings by looking them up in a table. This can allow you to translate data for use in systems that may have a more restricted character set.

The \$CHARMAP function uses the same table format as the \$MAP command, but differs in that it operates on one character of the input string at a time. The values in the left hand column of the table (i.e. before the separating tab) must be single characters, whereas the values in the right hand column can be any string. This command operates by taking each character in succession from the input string, matching it against a character in the left hand column, and replacing it by the corresponding string in the right hand column. If a match is not found, the input character is passed through to the output without modification. That is, only characters that are enumerated in the left column will be modified.

Note these additional points regarding the \$CHARMAP command:

- If a string with more than one character appears in the left column, only the first character is used as a match value.
- If an empty string appears in the left column (i.e. a tab or line terminator is found immediately), it is ignored.
- If you want to match with unprintable ASCII characters, you can use the ^ notation described in “Control and Escape Characters” on page 16.
- Special characters like backslash "\" or the open comment "{" can be mapped by preceding them with a backslash.

➤ See also: \$MAP \$VERIFY

Keyword	17 \$CHECK
Status	
Synopsis	\$CHECK(<i>Message</i>) \$DEVICES \$SIGNALS
Returns	N.A.
Type	Data
Where	Script – Device, Signal
Description	This command returns an alert box if the current list of devices or signals is not empty after a \$FIND command.
Examples	\$FIND \$DEVICES \$CHECK(The current circuit is not empty) \$DEVICES Will return an alert box if the circuit contains devices.

Keyword	18 \$CHECKSUM
Status	
Synopsis	\$CHECKSUM
Returns	Decimal integer.
Type	Data
Where	Script – Device
Description	Returns the checksum of the device type associated with the current object.
Examples	\$DEVICES\$DEVNAME \$CHECKSUM Will generate a listing of all devices in a circuit with their associated checksum.

The checksum of a device type is a program-generated 32-bit value which is generated from a random number each time a device type (i.e. a library symbol definition) is created or modified. It is used to distinguish between different versions of a symbol with the same name. This value is not normally of interest to the user, but it can be useful for error checking or locating cases where two device types of the same name have been used in a design inadvertently.

Keyword	19 \$CHILDEMTPPHASE
Status	
Synopsis	\$CHILDEMTPPHASE
Returns	Single character 'a' (phase-A signal), 'b' (phase-B signal), 'c' (phase-C signal), '3' (3-phase signal), 'X' (mismatch) or null when standard signal
Type	Data
Where	Script - Pin
Description	Applies only to pins on devices that have a subcircuit. Provides an indication of the type of power signal that is associated with the corresponding pin in the subcircuit .
Example	\$DEVPINFORMAT \$IF(\$EQ(\$CHILDEMTPPHASE,3))\$ALERT1(It's 3- phase)\$END \$DEVICES \$DEVNAME \$PINS

See also \$EMTPPHASE.

Keyword	20 \$CHILDSIGNAME
Status	
Synopsis	\$CHILDSIGNAME
Returns	Text
Type	Data
Where	Script – Pin
Description	If the current pin is associated with a device that has a subcircuit, then this keyword returns the name of the signal attached to the corresponding port in the subcircuit. If the device has no subcircuit, then this simply returns the name of the pin.
Examples	\$DEVPINFORMAT \$CHILDSIGNAME \$DEVICES \$TYPENAME \$PINS

Keyword	21 \$CIRCUITNAME
Status	
Synopsis	\$CIRCUITNAME
Returns	Text
Type	Data
Where	Script - Any object
Description	This returns the name of the circuit associated with the current object.
Examples	\$DEVICES \$DEVNAME \$CIRCUITNAME

This keyword is intended for use in hierarchical designs to get the name of the circuit associated with the current object. The following rules determine the value that is returned:

- If the current object is itself, or is inside, the top-level circuit in a hierarchical design, then this is the design name and is the same as \$DESIGNNAME.
- If the circuit is a subcircuit and the design is in Physical hierarchy mode, then the hierarchical path to the circuit is returned.
- If the circuit is a subcircuit, it is not open for editing, and the design is in Pure mode, the a name derived from the type name is returned.
- If the circuit is a subcircuit that is open for editing, and the design is in Pure mode, a complete path is returned, as in Physical mode.

Keyword	22 \$CLEARERRORBIT
Status	
Synopsis	\$CLEARERRORBIT(<i>bitNum</i>)
Returns	N. A.
Type	Action
Where	Script - Any object
Description	Clears the given bit number in the binary error set represented in the object's "OKErrors" attribute field. OKErrors attribute data format is hexadecimal.
Examples	<pre>\$DEVICES\$IF(\$ALERT2(Remove "Mark as OK" setting in \$DEVNAME?))\$CLEARERRORBIT(7)\$END</pre> <p>Will prompt the user if it's OK to remove the "Mark as OK" setting on the each device.</p>

This call is one of a set of commands designed to implement a "Mark as OK" feature in error checking scripts.

See the description of this feature under "Implementing Mark as OK in Error Checking Scripts" on page 28.

➤ See also: \$SETERRORBIT \$ERRORBITON \$ERRORBITOFF \$CLEARERRORS

Keyword	23 \$CLEARERRORS
Status	
Synopsis	\$CLEARERRORS
Returns	N. A.
Type	Action
Where	Script - Any object
Description	Sets the object's "OKErrors" attribute field to null.
Examples	<pre>\$DEVICES\$CLEARERRORS</pre> <p>Will remove all "Mark as OK" settings in all devices.</p>

This call is one of a set of commands designed to implement a "Mark as OK" feature in error checking scripts.

See the description of this feature under "Implementing Mark as OK in Error Checking Scripts" on page 28.

➤ See also: \$SETERRORBIT \$ERRORBITON \$ERRORBITOFF \$CLEARERRORBIT

Keyword	24 \$CLOSECIRCUIT/\$CLOSEDESIGN
Status	
Synopsis	\$CLOSECIRCUIT \$CLOSEDESIGN
Returns	N. A.
Type	Action
Where	Top level – design
Description	Closes the current design, exactly as if the user had selected the Close Design menu command. Note that this command closes the design without asking for saving design.
Examples	\$CLOSEDESIGN

Keyword	25 \$CLOSEREPORT
Status	
Synopsis	\$CLOSEREPORT
Returns	N. A.
Type	Action
Where	Top level
Description	Closes the current report file.
Examples	\$CLOSEREPORT

The \$CLOSEREPORT command causes the current report output file to be closed. This can be followed by another \$CREATEREPORT command to create another output file, thus allowing a single script file to generate multiple report files. The output file is normally closed automatically at the end of the script file, so this command is only necessary to create multiple output files. If \$CLOSEREPORT is not followed by a \$CREATEREPORT command, and subsequent listing commands attempt to write output data, then the user will be prompted to provide a name for a new output file.

Note that report files can be "nested" to any desired depth. I.e. if you do multiple successive \$CREATEREPORT commands without intervening \$CLOSEREPORTs, output will be directed to the file created by the most recent one. When it is closed, subsequent output will be directed to the next most recent one, etc.

Keyword	26 \$CLOSETRANSSCRIPT
Status	
Synopsis	\$CLOSETRANSSCRIPT
Returns	N. A.
Type	Action
Where	Top level
Description	Closes the current transcript file.
Examples	\$CLOSETRANSSCRIPT

This command closes the current transcript file, or does nothing if no transcript file is open. If no transcript file remains open, any subsequent uses of \$WRITETRANSSCRIPT will have no effect until another file is created using \$CREATETRANSSCRIPT.

Note that transcript files (like normal output files) can be "nested" to any desired depth. I.e. if you do multiple successive \$CREATETRANSSCRIPT commands without intervening \$CLOSETRANSSCRIPTs, output will be directed to the file created by the most recent one. When it is closed, subsequent output will be directed to the next most recent one, etc.

➤ See also: \$CREATETRANSSCRIPT.

Keyword	27 \$COL
Status	
Synopsis	\$COL(<i>n</i>)
Returns	Text
Type	Data
Where	Script
Description	If the current position in a line is less than N spaces from the left hand end then blanks are inserted until the Nth column is reached. If the output is already at or past the Nth column, nothing is output. For instance \$COL(20) will force any output to be indented to the 20th column of each line.
Examples	Name\$COL(20)Part Type\$COL(40)Package Code \$DEVICES\$DEVNAME\$COL(20)\$TYPENAME\$COL(40)&Package Will generate a simple bill of materials with items aligned at column 20 and 40.

Note that the column alignment generated by \$COL(*n*) is strictly based on character count, and so will depend on using a fixed-space font to generate printed output that is correctly aligned. If you intend to transfer the data to a word processor or spreadsheet for printing, it will probably be more appropriate to use tab separators between columns.

See also \$TAB.

Keyword	28 \$COMBDEVSON/\$COMBDEVSOFF
Status	
Synopsis	\$COMBDEVSON \$COMBDEVSOFF
Returns	N. A.
Type	Definition
Where	Top-Level
Description	When ON, causes any subsequent \$DEVICES listing to merge all devices with the same sort value onto one line of output. The default value is OFF. If a \$DEVICES listing is requested with \$COMBDEVSON and no sort, a warning message will be issued.
Examples	<pre>\$SORT \$DEVICES &Part \$COMBDEVSON \$DEVICES&Part\$TAB\$DEVNAME</pre> <p>This will produce a simple bill of materials with each line showing the part name followed by a listing of the devices with that part type. If \$COMBDEVSON was not used, the output would contain one line per device with only a single name on each line.</p>

More information on sorting and merging can be found in “Sorting and Merging” on page 23.

Keyword	29 \$COMBPINSON/\$COMBPINSOFF
Status	
Synopsis	\$COMBPINSON \$COMBPINSOFF
Returns	N. A.
Type	Definition
Where	Top-Level
Description	<p>When ON causes multiple pin connections on the same device to to be combined without repeating the device name. The default is OFF. For example:</p> <p>With \$COMBPINSON: IC1-2,5,6,12 With \$COMBPINSOFF: IC1-2 IC1-5 IC1-6 IC1-12</p>
Examples	\$COMBPINSON

Keyword	30 \$COMBSIGSON/\$COMBSIGSOFF
Status	
Synopsis	\$COMBSIGSON \$COMBSIGSOFF
Returns	N. A.
Type	Definition
Where	Top-Level
Description	When ON, causes any subsequent \$SIGNALS listing to merge all signals with the same sort value onto one line of output. The default value is OFF. If a \$SIGNALS listing is requested with \$COMBSIGSON and no sort, a warning message will be issued.
Examples	\$SORT \$SIGNALS \$PAGENUM \$COMBSIGSON \$SIGNALS\$PAGENUM\$TAB\$SIGNAME This will produce a list of signals by page number with one line per page.

More information on sorting and merging can be found in “Sorting and Merging” on page 23.

Keyword	31 \$CONTAINS
Status	
Synopsis	\$CONTAINS(string1, string2)
Returns	Boolean
Type	Definition
Where	Top-Level
Description	This command returns TRUE if string1 contains string2
Examples	\$DEVICES \$CONTAINS(&Name,R)

Keyword	32 \$CONTEND
Status	
Synopsis	\$CONTEND(<i>string</i>)
Returns	N. A.
Type	Definition
Where	Top-Level
Description	This command specifies a string to be added at the end of any line that will be continued on the next line due to the line width or item count being exceeded.
Examples	\$LINEWIDTH(10) \$CONTEND(+)

➤ See also: \$CONTSTART

Keyword	33 \$CONTSTART
Status	
Synopsis	\$CONTSTART(<i>string</i>)
Returns	N. A.
Type	Definition
Where	Top-Level
Description	This command specifies a string to be inserted at the start of any continuation line generated due to the line width or item count being exceeded.
Examples	<p>\$CONTSTART(+)</p> <p>This will insert a "+" continuation character, typical of SPICE-based netlist formats.</p>

See also: \$CONTEND

Keyword	34 \$COUNT
Status	
Synopsis	\$COUNT
Returns	Decimal integer
Type	Data
Where	\$DEVICES or \$SIGNALS listing
Description	This keyword returns the number of different Name values held by devices or signals merged on the current line. This is intended for use in flat PCB designs in which the package assignment is stored in the Name field. In this case \$COUNT in effect returns the number of physical devices represented by this line.
Examples	<p>\$SORT \$DEVICES &Part \$COMBDEVSON \$DEVICES&Part \$COUNT</p> <p>This will produce a listing showing the number of each part type required by the current design. Devices with multiple gates per package will count as 1 because they have the same value in Name.</p>

For a corresponding count in Physical hierarchy designs, see \$COUNTINST.

Keyword	35 \$COUNTINST
Status	
Synopsis	\$COUNTINST
Returns	Decimal integer
Type	Data
Where	\$DEVICES or \$SIGNALS listing
Description	This keyword returns the number of different InstName values held by devices or signals merged on the current line. This is intended for use in physical mode PCB designs in which the package assignment is stored in the InstName field. In this case \$COUNTINST in effect returns the number of physical devices represented by this line.
Examples	<pre>\$SORT \$DEVICES &Part \$COMBDEVSON \$DEVICES&Part \$COUNTINST</pre> <p>This will produce a listing showing the number of each part type required by the current design. Devices with multiple gates per package will count as 1 because they have the same value in InstName.</p>

For a corresponding count in Flat mode designs, see \$COUNT.

Keyword	36 \$COUNTVALUES
Status	
Synopsis	\$COUNTVALUES(<i>string</i>)
Returns	Decimal integer.
Type	Data
Where	Script - any object
Description	When multiple objects with the same sort value are merged on the same line, any object-based value, such as \$DEVNAME, could generate multiple different values. This command provides a count of the number of values generated by the command string given as its argument.
Examples	<pre>\$SORT \$DEVICES &Part \$COMBDEVSON \$DEVICES&Part \$COUNTVALUES(\$DEVNAME)</pre> <p>This script will list all part types in the circuit with a count of the number of different usages of each type. This takes into account the fact that multiple symbols may be assigned to one package, and will therefore have the same name.</p>

This command is used in conjunction with the sorting and merging features of the Scripter to count the number of different values of a field found in a collection of objects. It can be used for error-checking purposes to ensure that some group of objects have all been assigned the same value, or can be used to generate counts of various kinds in parts lists.

See also: \$SORT \$COMBDEVSON/\$COMBDEVSOFF \$SINGLE \$MERGE

Keyword	37 \$CREATEFOLDER/\$CREATEDIRECTORY
Status	
Synopsis	\$CREATEFOLDER(<i>dirName</i>) \$CREATEDIRECTORY(<i>dirName</i>)
Returns	N. A.
Type	Action
Where	Top level
Description	Creates a directory with the given name. If no path is given, the directory will be created inside the "current" directory, usually the one containing the current design. \$CREATEFOLDER and \$CREATEDIRECTORY are identical in function.
Examples	\$CREATEDIRECTORY(\$DESIGNNAME Reports)

This command creates a directory on disk. This is intended specifically for cases where netlist formats require the generation of multiple files and it is most convenient to place them in a separate directory.

The argument string can contain only the name for the new directory or it can specify a relative or absolute path name. If no path is given, the new directory will be created in the current directory. This is usually the directory containing the current design, unless there is no current design or it has been set by another command.

If a relative path is given, it is relative to the current directory. This command will only create the bottommost directory in the path, the others must already exist.

If an absolute path is given, it must start with the disk name.

If the specified directory already exists, it is left untouched and no error is given. After this command has completed successfully, the current directory will be the newly-created one (or the pre-existing one, if it was already there).

See more information about directories and path names in "File Names and Paths" on page 37.

See also : \$FOLDER \$DIRECTORY

Keyword	38 \$CREATEREPORT
Status	
Synopsis	\$CREATEREPORT[(nameString)][\$PROMPT] \$CREATEREPORT \$NULL
Returns	N. A.
Type	Action
Where	Top level
Description	Creates a report output file.
Examples	\$CREATEREPORT(\$DESIGNNAME.net) \$PROMPT

This command creates a text file for subsequent report output. This allows you to eliminate or control the automatic file save prompt that occurs when a script generates any output.

Note that output files can be "nested" to any desired depth. I.e. if you do multiple successive \$CREATEREPORT commands without intervening \$CLOSEREPORT, output will be directed to the file created by the most recent one. When it is closed, subsequent output will be directed to the next most recent one, etc.

If the \$PROMPT option is specified, the user will be prompted with a standard file save box with the given name string as the default file name.

If \$PROMPT is not specified, then the file will be created immediately with no user prompt. If the file already exists, it is replaced with no user prompt. The file is created in the current directory, which is normally the one containing the current design file. Note that \$CREATEREPORT does not accept a pathname as part of the file name. The directory can be specified by using \$FOLDER or \$DIRECTORY to change the current directory.

The \$NULL option causes a "null" file to be created, i.e. all output generated by the script will be discarded. It cannot be used in conjunction with any of the other options.

For more information on text output files, see "File Input and Output" on page 37.

➤ See also: \$CLOSEREPORT \$CREATETRANSSCRIPT

Keyword	39 \$CREATETRANSSCRIPT
Status	
Synopsis	\$CREATETRANSSCRIPT[(<i>nameString</i>)] [\$PROMPT]
Returns	N. A.
Type	Action
Where	Top level
Description	Creates a transcript output file.
Examples	\$CREATETRANSSCRIPT(\$DESIGNNAME Error Log)

This command creates a text file for subsequent transcript output, i.e. any strings written with the \$WRITETRANSSCRIPT command. In terms of format and options, this command is identical to \$CREATEREPORT.

For more information on text output files, see “File Input and Output” on page 37.

See also: \$WRITETRANSSCRIPT \$CREATEREPORT

Keyword	40 \$DATE
Status	
Synopsis	\$DATE \$DATE(formatString) \$DATE(formatString,valueString)
Returns	Text
Type	Data
Where	Script
Description	This command is used to display the current date or to convert raw decimal integer date/time values to a desired format. The first form returns the current date in the default format. The second form is used to specify any time or date format using format keywords. The third form is used to provide a raw date value for conversion. When used with an argument list, \$TIME and \$DATE are identical in function.
Example	This report was produced on a \$DATE(\$W) This will generate the day of the week, fully spelled out.

When used without an argument list, \$DATE generates the current date in the default "long" format for the host machine. This behaviour can be modified by adding an argument string containing format keywords for the various date and time elements that are available. Any characters in the format string that are not recognized as one of the following items will be included literally in the output string. If a \$ character is needed in the output, it can be escaped by preceeding it with a backslash.

If a second argument is provided, it is assumed to be a decimal integer representing a date/ time in raw form. This format is used to store dates for a variety of internal purposes such as device date stamping, file modified dates, etc., so \$DATE can be used to convert them for output.

See the table of date and time codes under \$TIME.

➤ See also: \$TIME \$DATECREATED/\$DATEMODIFIED

Keyword	41 \$DATECREATED/\$DATEMODIFIED
Status	
Synopsis	\$DATECREATED \$DATEMODIFIED \$DATECREATED(formatString) \$DATEMODIFIED(formatString)
Returns	Text
Type	Data
Where	Script
Description	If used without arguments, \$DATECREATED and \$DATEMODIFIED return the created or modified date of the current design in the default format. An argument list can be added to specify any time or date format. When used with an argument list, \$TIMECREATED and \$DATECREATED are identical in function and, similarly, \$TIMEMODIFIED and \$DATEMODIFIED are identical.
Example	Design \$DESIGNNAME was created on \$DATECREATED

These two commands are variations of the \$DATE command and behave identically except that they use the created or modified date of the current design, rather than the current date.

➤ See also \$DATE \$TIMECREATED/\$TIMEMODIFIED

Keyword	42 \$DEFINEATTR
Status	
Synopsis	\$DEFINEATTR(fieldName/options[maxLength])
Returns	N.A.
Type	Action
Where	Top level
Description	Used to define a new attribute field for the design or modify settings in an existing one.
Example	\$DEFINEATTR(TraceWidth/SY) This will define a field called TraceWidth for signals. It will be marked as Primary. If the field already exists, it will be set to Primary if it is not already.

This command allows you to define an attribute field in the current design. It is not strictly necessary to use this command to define a field before using it in a script. A \$SETATTR on such a field will always define it automatically with default settings in the design's attribute table. However, the default settings may not be the desired ones and may change in future versions of EMTWorks. For this reason, it is preferable to define a field before setting it. This way, you can ensure that settings like the maximum length and the various options are appropriate for its intended usage.

The \$DEFINEATTR command can also be used to change certain settings in existing fields in a design. No changes are allowed that would affect the usage of any existing fields. The items that can be changed are noted in the table below.

The format string that is used to define a new field consists of:

- The name of the field
- A slash character "/"

- A number of upper case letters, each of which indicates an option. At least one option must be specified, that being the allowable object type, i.e. device, signal, pin or design.
- An optional decimal integer indicating the maximum length.

Note that when attribute data is stored internally, it is always allocated as a variable-length string, so there is no wastage of memory space by specifying a longer maximum length. The default value is the maximum value of 32,000 characters.

The following table defines the meaning of the format code letters. Note that letters must be given in upper case.

\$DEFINEATTR Format Codes

Format Letter	Modify Existing Fields?	Meaning
D	No	Device
S	No	Signal
P	No	Pin
C	No	Design/Circuit
V	Yes	Visible by default
F	No	Fixed (i.e. cannot be changed by the user)
X	No	Value fixed - i.e. read only
I	No	Keep with instance
R	Yes	Rotate with object
N	No	Name characters only - not implemented
G	Yes - except Name and InstName	Group with name
W	Yes	Show field name
A	No	Allow carriage returns
L	Yes	Location fixed
Y	Yes	In primary list
T	No	Link to Part
U	No	Numeric characters only - not implemented
M	No	Temporary field - not saved with file, not visible to user

At least one of D, S, P or C must be specified.

See also \$SETATTR and the general information on attribute definitions provided in chapter "Attributes" in the EMTPWorks User's Guide.

Keyword	43 \$DEFINEBLOCK
Status	.
Synopsis	\$DEFINEBLOCK(<i>blockName</i>)
Returns	N.A.
Type	Definition
Where	Top level
Description	Indicates the start of a block within a script.
Examples	<pre>\$DEFINEBLOCK(InternalCct) \$DEVICES\$DEVNAME \$PINS \$END \$DEVICES\$INTERNAL(InternalCct)</pre> <p>This example creates a block that is referred to by the \$INTERNAL command. I.e. The script lines in the block will be executed to output the internal circuit of each device listed in the last line.</p>

A block is simply a contiguous sequence of lines within a script file. The \$DEFINEBLOCK command indicates the start of a block, but does not impose any meaning on it. The interpretation of the contents of the block depends entirely on the command that refers to it. Some commands, such as \$INTERNAL, \$INCLUDE or \$EXECUTE, treat the block as a sort of "subroutine", that is, a sequence of commands to be executed when called upon. Other commands, such as \$MAP, treat the block as a table of literal text values.

The only rules about the contents of a block are imposed by the requirement that the script parser must be able to skip over the contents and find the corresponding \$END. This imposes the following limits on block format:

If the block contains commands, they must be a complete and valid set of commands. In particular, any structure that requires an \$END (e.g. \$IF) must be completed within the block.

If the block contains literal data, there must be no data items that might confuse the parser in its search for an \$END. In particular, any items starting with a \$ character must be escaped with a backslash.

When the Scripter encounters a \$DEFINEBLOCK command in the file, it notes the starting point of the block and then skips over it. Execution resumes on the next line past the \$END.

➤ See also: \$INCLUDE \$INTERNAL \$MAP \$VERIFY \$TABLE \$INLINE

Keyword	44 \$DEFINECIRCUIT
Status	.
Synopsis	\$DEFINECIRCUIT
Returns	N.A.
Type	Definition
Where	Top level
Description	Indicates the start of an internal circuit definition within a script. This is actually a special case of \$DEFINEBLOCK and is exactly equivalent to: \$DEFINEBLOCK(_Internal_)
Examples	<pre>\$DEFINECIRCUIT \$DEVICES\$DEVNAME \$PINS \$END \$DEVICES\$INTERNAL</pre> <p>This example creates a block that is referred to by the \$INTERNAL command. I.e. The script lines in the block will be executed to output the internal circuit of each device listed in the last line.</p>

Keyword	45 \$DESIGNNAME
Status	
Synopsis	\$DESIGNNAME
Returns	Text
Type	Data
Where	Script - design
Description	Returns the name of the current design, i.e. the file name with any "." extensions removed.
Examples	This report was generated from \$DESIGNNAME

\$DESIGNNAME can be used in the following areas:

- In an \$INCLUDE command, for generating the name of the include file.
- In a \$CREATEREPORT command, for generating the name of the report output file.
- In a \$HEADER section.
- Anywhere else in the text of a script that is not part of some other definition or listing command.

\$DESIGNNAME vs. \$FILENAME

The operation of the \$DESIGNNAME and \$FILENAME keywords is designed to make it easier to work with "." dot extensions for report files.

\$DESIGNNAME will be replaced by the name of the currently open design with any "." extension removed. I.e. The program starts at the end of the name and scans forward. If a "." is found, it and all following characters are removed. If no "." is found, the name is not changed.

\$FILENAME returns the name of the current design file verbatim, including any "." extension.

\$DESIGNNAME vs. \$CIRCUITNAME

In a flat (that is, non-hierarchical) design, \$DESIGNNAME and \$CIRCUITNAME are equivalent. If the current object is, or is contained in, a sub-circuit in a hierarchical design, \$DESIGNNAME will always refer to the name of the topmost circuit, whereas \$CIRCUITNAME gives a hierarchical name of the current object's circuit.

➤ See also: \$CIRCUITNAME \$CREATEREPORT \$INCLUDE

Keyword	46 \$DESIGNPATH
Status	
Synopsis	\$DESIGNPATH
Returns	Text
Type	Data
Where	Script - design
Description	Returns the directory path to the current design file. If the current design file has not been saved, this returns a null string. This behaviour provides a way of checking if the design has been saved.
Examples	The design is in directory \$DESIGNPATH

This command provides the complete file path to the current design file, not including the file name itself.

File paths start with the disk letter/colon or server name, followed by the directory path with backslash characters as a separator. For example:

```
d:\projects\controller board\
```

Care must be taken in specifying path names in a report file. The backslash character used as a directory separator is also the script language's escape character. For this reason, if a file path name must be specified explicitly in a script, you will have to double the backslash characters, as in this example:

```
$SETVAR(SysDir, c:\\windows\\temp\\)
```

➤ See also: \$SCRIPTPATH \$TEMPPATH \$PROGPATH \$FULLPATH

Keyword	47 \$DESIGNPINSIGSOURCE/\$DESIGNSIGSOURCE
Status	
Synopsis	\$DESIGNSIGSOURCE & <i>fieldName</i> \$DESIGNPINSIGSOURCE & <i>fieldName</i>
Returns	N. A.
Type	Definition
Where	Top level
Description	Specifies a design attribute which will contain a list of device attribute fields to be used as signal sources.
Examples	\$DESIGNSIGSOURCE &SigSources

The \$DESIGNSIGSOURCE and \$DESIGNPINSIGSOURCE commands allow a single netlist script to be used for multiple different designs that may have different ground and power requirements.

In each case, the name given is the name of a design attribute field which will contain a list of attribute fields to search for pins. I.e. this is equivalent to specifying a \$SIGSOURCE or \$PINSIGSOURCE command for each field listed in the design attribute.

It is also valid to use these commands in conjunction with the regular \$SIGSOURCE or \$PINSIGSOURCE commands. For example, your script could use \$SIGSOURCE commands to list the standard Power and Ground fields and then add the line:

```
$DESIGNSIGSOURCE &ExtraFields
```

to allow individual designers to add fields for their specific requirements. In this case, for example, placing the value:

```
AnaGnd,AnaPlus12,AnaMinus12
```

in the design attribute ExtraFields would have the same effect as placing the following three commands in the script file:

```
$SIGSOURCE &AnaGnd
$SIGSOURCE &AnaPlus12
$SIGSOURCE &AnaMinus12
```

Some additional notes on the usage of this command:

- In the current implementation of the \$DESIGNSIGSOURCE command, there is no way to specify one name for the attribute field and a different name for the net name that appears in the netlist. This can only be done using the \$SIGSOURCE command explicitly for each name.
- The presence of a \$DESIGNSIGSOURCE or \$DESIGNPINSIGSOURCE command in the script does not add any significant execution time overhead if no values are specified in the given design attribute field. If a value is specified, the note given under the \$PINSIGSOURCE command will apply.

➤ See also: \$SIGSOURCE \$PINSIGSOURCE

Keyword	48 \$DEVCOUNT
Status	
Synopsis	\$DEVCOUNT
Returns	Decimal integer
Type	Data
Where	Top level
Description	Returns the number of lines in the last \$DEVICES listing. If \$COMBDEVSON has been set, this will mean in effect the number of different sort values in the list.
Examples	<pre>\$COMBDEVSON \$SORT \$DEVICES &Part \$DEVICES&Part \$DEVNAME Number of different part types: \$DEVCOUNT</pre> <p>This will generate a simple bill of materials with one line per part type and a final line indicating the number of different part types, i.e. the number of unique values in the Part attribute field.</p>

\$DEVCOUNT is used to get a count of the number of devices on the scripter's current device list. It is only valid after one of the following types of statements has been executed:

```
$DEVICES
$FIND $DEVICES
$SORT $DEVICES
```

If a \$SORT has been done and \$COMBDEVSON has been set, \$DEVCOUNT returns the number of unique sort values in the devices list, otherwise it returns the number of devices in the list. In other words, it returns the number of lines that would be generated by a \$DEVICES listing.

For more information on sorting, see "Sorting and Merging" on page 23.

Keyword	\$DEVHIERNAME
Status	
Synopsis	\$DEVHIERNAME
Returns	Text
Type	Data
Where	Script – device
Description	Returns the hierarchical name of the current device, i.e. the device name prefixed with the names of any parent devices in the hierarchy. In Flat designs, this is equivalent to \$DEVNAME. This is not valid in Pure designs since there is no unequivocal path from a given device to the top-level circuit.
Examples	<pre>\$DEVICES\$DEVHIERNAME \$PINS</pre> <p>This will generate a list of devices showing the hierarchical name of each followed by a list of the attached pins.</p>

\$DEVHIERNAME is intended to provide a unique device identifier when generating flattened netlists from hierarchical designs. Here are some important points to remember when using this keyword:

The hierarchical name consists of the Name value for the current device with the "path" formed by its parent devices prefixed to it, such as:

MEMBLK1/CONTROL/CTR2

The separator character that is used in generating these names can be set using the \$HIERNAMESeparator command.

Hierarchical names can be arbitrarily long, depending on the nesting level. For this reason, they may not be suitable as an identifier in many netlist formats.

- See also \$HIERNAMESeparator and "Script Hierarchy Issues" on page 32.

Keyword	49 \$DEVICES
Status	
Synopsis	\$DEVICES formatItems
Returns	Text
Type	Data
Where	Top level
Description	Creates a listing of devices using the format items following on the line.
Examples	\$DEVICES\$DEVNAME \$PINS This will generate a list of devices showing the name of each followed by a list of the attached pins.

The \$DEVICES keyword also appears as a modifier of the \$FIND, \$SORT, \$ASSIGNNAMES, \$ASSIGNINSTNAMES and \$BREAK commands. See the entries for those commands for more information.

The \$DEVICES command is the primary command for generating any listing by device. The format of the listing is determined by the format items that follow the command on the line and by the various definition options, as outlined here:

- The scope of the listing in hierarchical designs is affected by the \$HIERARCHY command.
- A subset of the devices in the design can be extracted using any desired criteria using the \$FIND command.
- The ordering of devices in the listing is affected by the \$SORT command.
- If the \$COMBDEVSON has been selected, multiple items with the same sort value will appear on one line.
- The \$NEWLINE format command can force multiple lines to be generated for each item.
- Limits on line length imposed by the \$LINEWIDTH and \$MAXITEMSPERLINE commands may cause multiple lines to be generated.
- The format of the listing for each device or signal is completely determined by the text and commands which follow the \$DEVICES or \$SIGNALS keyword. There is no default format, so if no line format is specified, a sequence of blank lines will be written to the file.
- EMTPWorks "pseudo-devices" such as page connectors, signal connectors and bus breakouts are normally not included in any listing.
- Any characters appearing after \$DEVICES that is not part of a format sub-command will be placed verbatim in each line of the listing. For example, blanks, tabs or commas can be used to format each line.

Numerous data-generating commands can be used in defining the output generated on each line. Any command listed in this Appendix with a "Where" value of "Script - Device", "Script - Circuit", "Script - Any object" or "Script" can be used on a \$DEVICES line. Some of the more common format items are summarized in this table. See the corresponding keyword section for each item for more information.

\$DEVNAME	The name of the device.
\$DEVSEQ	The sequence number of the current device, i.e. an integer assigned to the device, starting at "n" for the first device in the sorted sequence and incremented for each subsequent one.
\$TYPENAME	The device's type name, that is, the name used to refer to this type in the library.
\$COUNT	The number of devices being merged to form this line, i.e. the number having the same sort value.
\$SINGLE	Tells the report generator to write only a single value for all items following on the line, regardless of how many items were combined with the same sort value.
\$NUMPINS	The number of pins on the device.
\$PAGE	The circuit page that this device appears on.
\$PINS	A list of the device's pins. The format of each item is determined by the \$DEVPINFORMAT command.
\$COL(N)	If the current position in a line is less than N spaces from the left hand end then blanks are inserted until the Nth column is reached.
\$NEWLINE	Writes a new line character into the file. This allows a device entry to occupy several lines in the output report.
&attr	Inserts the value of the attribute field specified.

A variety of applications for the \$DEVICES command are described in "Script Examples" on page 44.

Keyword	50 \$DEVINSTNAME
Status	
Synopsis	\$DEVINSTNAME
Returns	Text
Type	Data
Where	Script - device
Description	Returns the instance name of the current device, i.e. the contents of its InstName attribute field. This field is normally used to hold the device package assignment in Physical hierarchy mode designs.
Examples	<p>\$DEVICES\$DEVINSTNAME \$PINS</p> <p>This will generate a list of devices showing the instance name of each followed by a list of the attached pins.</p>

Keyword	51 \$DEVLOC
Status	
Synopsis	\$DEVLOC
Returns	Text
Type	Data
Where	Script - device
Description	Returns the hierarchical locator string for the current device.
Examples	\$DEVICES\$DEVNAME \$DEVLOC

The \$DEVLOC command generates a unique identifier string referred to as a "locator". This can be used to uniquely identify any device within a hierarchical design, even if some devices have duplicate or null names. The locator is used by other EMTPWorks modules like ErrorScript to locate objects unambiguously.

Keyword	52 \$DEVNAME
Status	
Synopsis	\$DEVNAME
Returns	Text
Type	Data
Where	Script - device
Description	Returns the name of the current device, i.e. the contents of its Name attribute field.
Examples	<p>\$DEVICES\$DEVNAME \$PINS</p> <p>This will generate a list of devices showing the name of each followed by a list of the attached pins.</p>

In Flat mode PCB designs, \$DEVNAME is normally used for the device package assignment. In Physical and Pure mode hierarchical designs, \$DEVNAME is the "reference designator", or a logical name that is unique to each symbol within a circuit level, but not necessarily unique in the entire design.

For more information on names in hierarchical designs, see \$HIERARCHY on page 98.

Keyword	53 \$DEVPINFORMAT
Status	
Synopsis	\$DEVPINFORMAT <i>formatItems</i> \$DEVPINFORMAT(<i>formatItems</i>)
Returns	Text
Type	Data
Where	First form - Top level Second form - Script - device
Description	Sets the format that will be used for each pin output by the next \$PINS command in a \$DEVICES line. The first form (without parentheses) can only be used on a top-level line. The second form (with parentheses) can be used either on a top-level line or in a \$DEVICES line. This allows the format to be modified on the fly while generating output for a single device. This is required in some netlist formats that require a pin list header followed by a declaration list.
Examples	<pre>\$DEVPINFORMAT [\$PINNAME,\$PINNUM] \$ITEMSEPARATOR(,) \$DEVICES\$DEVNAME \$PINS</pre> <p>This will generate a list of devices showing the name of each followed by a list of the attached pins using the format specified:</p> <pre>U1 [A,1],[B,2],[Q,3]</pre>

This command sets the format for pin lists (i.e. a \$PINS command) appearing in a \$DEVICES listing. The default format is "\$\$SIGNAME", meaning the signal name alone will appear for each device pin.

Here are some notes on device pin formats:

- Leading and trailing blanks are always trimmed from the format list, but embedded blanks (i.e. those surrounded by non-blank format items) are not. If you want a leading or trailing blank in a format, it must be preceded by an escape character (backslash).
- Any literal text not recognized as a command or terminator will be placed in the format verbatim.
- If you need to include parentheses in the pin format, they should be preceded by an escape character to ensure that they are not interpreted as argument terminators.
- Any command that is valid for a device, signal, pin or design can be used in a pin format.

If an attribute reference appears in a pin format, there can be some ambiguity since it can refer to a device, signal or pin. See "Precedence of Field References in Pin Listings" on page 18 for a definition of the search order used by the Scripter in evaluating these references.

Keyword	54 \$DEVPINSEQUENCE
Status	
Synopsis	\$DEVPINSEQUENCE &attrField
Returns	N. A.
Type	Definition
Where	Top level
Description	Sets an attribute field to be searched for a pin order specification while creating a \$PINS listing.
Examples	<p>\$DEVPINSEQUENCE &PinSequence</p> <p>This specifies that the standard field PinSequence is to be searched for a list of pins to output.</p>

Some design programs, such as SPICE-based simulators, require netlist formats in which the device pin order is significant. Since there are no standards for pin order, different packages will likely require different ordering schemes. In order to allow a variety of pin orderings to be stored with each device type, the Scriptor tool can be asked to search a device attribute field for pin order information. The name of this field is specified in the \$DEVPINSEQUENCE command, as follows:

```
$DEVPINSEQUENCE &attrField
```

Once this field has been specified, if a \$PINS sub-command is encountered in a \$DEVICES listing, each device is searched for an attribute field with the given name. If the field is found (in the device instance or the type attributes), then it is assumed to contain a list of names of the pins to write out to the file. The pin names must be listed exactly as they appear in the “Edit symbol” option pin list, or, in the case of \$SIGSOURCE pins, as the fields are named in the device attributes. If the specified field is not found in a given device, then the normal pin ordering is used.

For example, assume a 2-input gate has pins named A, B and OUT. A pin sequence attribute field could be specified in the device (or in the library type attributes) as follows:

```
OUT, A, B
```

The command:

```
$DEVPINSEQUENCE &PinSequence
```

in the script would then cause subsequent listings of that device to have pins appear in the specified order. The predefined field PinSequence is intended for this purpose.

➤ See also: \$PINS

Keyword	55 \$DEVSEQ
Status	
Synopsis	\$DEVSEQ[(<i>origin</i>)]
Returns	Decimal integer
Type	Data
Where	Script - device
Description	Returns an integer indicating the position in the sort sequence of this device. If the original is not specified, numbering starts at zero.
Examples	<pre>\$SORT \$DEVICES \$DEVNAME \$DEVICES\$DEVSEQ(1). \$DEVNAME</pre> <p>This will generate a listing of devices that looks something the following:</p> <ol style="list-style-type: none"> 1. C1 2. C2 3. U1 4. U2

\$DEVSEQ returns the sequence number of the current device, that is, an integer assigned to the device, starting at "origin" for the first device in the sorted sequence and incremented for each subsequent one. This number is not permanently associated with the device but it strictly its sequence number in the current sort. The "(origin)" part is optional. If it is omitted the sequence starts at zero. If no sort has been done, all devices will have the origin sequence number.

Keyword	56 \$DEVTOKEN
Status	
Synopsis	\$DEVTOKEN
Returns	Decimal integer
Type	Data
Where	Script - device
Description	Returns an integer representing the device's token number.
Examples	<pre>\$DEVICES\$DEVNAME \$DEVTOKEN</pre> <p>This will generate a listing of devices with the name and token number of each.</p>

Keyword	57 \$DIRECTORY
Status	See \$FOLDER.
Synopsis	\$DIRECTORY(string)

Keyword	58 \$DIV
Status	
Synopsis	\$DIV(string1,string2)
Returns	Decimal integer
Type	Data
Where	Script
Description	Performs an integer division (first argument divided by second) and returns a decimal string of the result. If the second argument is zero, an error message is returned. All arithmetic is done with 32-bit signed values.
Example	\$SETVAR(Avg, \$DIV(&Total, \$COUNT)) Sets Avg variable value. It will be equal to (&Total/\$COUNT)

➤ See also: \$MINUS \$MULT \$PLUS

Keyword	59 \$DWVERSION
Status	
Synopsis	\$DWVERSION
Returns	Text
Type	Data
Where	Script
Description	Returns the version number of the EMTPWorks package it is running on.
Examples	* Generated by EMTPWorks \$DWVERSION

The keyword is used to insert the current EMTPWorks version number into output files for documentation purposes.

Keyword	60 \$ELSE
Status	See \$IF.

Keyword	61 \$EMTPPHASE
Status	
Synopsis	\$EMTPPHASE
Returns	Single character 'a' (phase-A signal), 'b' (phase-B signal), 'c' (phase-C signal), '3' (3-phase signal), 'X' (mismatch) or null when standard signal
Type	Data
Where	Script - Device, Pin or Signal
Description	Provides an indication of the of power signal the object is associated with.
Example	\$IF(\$EQ(\$EMTPPHASE,3))\$ALERT1(It's 3-phase)\$END

Keyword	62 \$END
Status	See \$IF, \$DEFINEBLOCK, \$DEFINECIRCUIT, \$HEADER

Keyword	63 \$EQ
Status	
Synopsis	\$EQ(string1,string2)
Returns	Boolean
Type	Data
Where	Script
Description	Performs an integer conversion on both arguments and returns TRUE if they are equal.
Example	\$IF(\$EQ(\$DATE(\$H),12))\$ALERT1(Time for lunch.)\$END

This is a numerical comparison, not a string comparison; for a string comparison, use \$REGEXP.

➤ See also: \$GE \$GT \$LE \$LT \$NE

Keyword	64 \$ERRORBITOFF
Status	
Synopsis	\$ERRORBITOFF(<i>bitNum</i>)
Returns	Boolean
Type	Data
Where	Script - Any object
Description	Tests if the given bit number in the binary error set represented in the object's "OKErrors" attribute field is off.
Examples	<p>\$DEVICES\$IF(\$AND(\$NOT(\$DEVNAME),\$ERRORBITOFF(7)))Error\$END</p> <p>Will output the text "Error" for each device that has no name and for which bit 7 of the OKErrors field is 0.</p>

This call is one of a set of commands designed to implement a "Mark as OK" feature in error checking scripts. See the description of this feature in "Implementing Mark as OK in Error Checking Scripts" on page 28.

➤ See also \$CLEARERRORBIT \$ERRORBITON \$SETERRORBIT

Keyword	65 \$ERRORBITON
Status	
Synopsis	\$ERRORBITON(<i>bitNum</i>)
Returns	"1" if the given error bit is 1.
Type	Data
Where	Script - Any object
Description	Tests if the given bit number in the binary error set represented in the object's "OKErrors" attribute field is on.
Examples	<pre>\$DEVICES\$IF(\$ERRORBITON(7))\$DEVNAME marked as OK!\$END</pre> <p>Will output the device name and the given string for each device that has bit 7 of the OKErrors field set.</p>

This call is one of a set of commands designed to implement a "Mark as OK" feature in error checking scripts. See the description of this feature in "Implementing Mark as OK in Error Checking Scripts" on page 28.

➤ See also: \$CLEARERRORBIT \$ERRORBITOFF \$SETERRORBIT

Keyword	66 \$EVAL
Status	
Synopsis	\$EVAL(<i>string</i>)
Returns	The result of processing the argument string a second time for script commands.
Type	Data
Where	Script - Any object
Description	Evaluates the argument string and then treats the result as a command string and evaluates it again.
Examples	<pre>\$SYSTEMOPEN(\$EVAL(&ARG1))</pre> <p>Takes the contents of variable ARG1 (presumably, a parameter passed by an outside invocation of the script) and re-evaluates it for commands. This allows the parameter passed in to contain more script commands, such as \$DESIGNNAME. If \$EVAL was not used here, the contents of the variable would be passed literally to \$SYSTEMOPEN.</p>

This call is used for cases where you want to read a line of script commands from another source and execute them as if they were part of the script.

➤ See also: \$SYSTEMOPEN

Keyword	67 \$FILEEXISTS
Status	
Synopsis	\$FILEEXISTS(<i>fileName</i>)
Returns	"1" if the given file exists, null otherwise.
Type	Data
Where	Script
Description	This command is used to determine if a specified file exists on disk. The file name can optionally include an absolute or relative directory path. If the directory is not absolutely specified, it is relative to the "current directory", which is usually the one containing the current design file.
Examples	<pre> \$IF(\$NOT(\$FILEEXISTS(\$DESIGNNAME.proj))) \$CREATEREPORT(\$DESIGNNAME.proj) Project file for \$DESIGNNAME, created \$DATE \$CLOSEREPORT \$END </pre>

- See also: \$FOLDER \$DESIGNPATH \$SCRIPTPATH \$TEMPPATH \$PROGPATH, and "File Names and Paths" on page 37.

Keyword	68 \$FILENAME
Status	
Synopsis	\$FILENAME
Returns	Text
Type	Data
Where	Script – design
Description	Returns the name of the current design's file. If the current design has never been saved to a file, this will be the name of the design window.
Examples	This report was generated from \$FILENAME

- See also: \$DESIGNNAME

Keyword	69 \$FIND
Status	
Synopsis	\$FIND [\$NOCLEAR] \$DEVICES[\$SIGNALS[criterion1][criterion2]...
Returns	N. A.
Type	Definition
Where	Top level - design
Description	Scans through the list of devices or signals in the current circuit or design to locate objects meeting the given criteria. After this command has executed, subsequent commands dealing with the same object type will see only the selected objects.
Examples	\$FIND \$DEVICES \$NOT(\$DEVNAME) This will locate all devices having no name, i.e. a null "Name" field.

This command is one of the most important and powerful ones in the scripting language. It allows you to selectively include or omit objects from consideration based on almost any criteria. For example, you can use it to locate:

- Devices of a certain type that require special formatting in a Netlist. E.g. Sub-circuit devices in a SPICE file.
- Devices whose pins meet some error condition (e.g. duplicate pin numbers)
- Only signals that have two or more pin connections for netlisting.
- Special device symbols that are used to mark the schematic or pass special parameters to an external system.
- Port connectors that will be used to defined a sub-circuit header in a hierarchical netlist.

In order to implement the \$DEVICES and \$SIGNALS commands, the Scripter keeps two internal lists, one for devices and one for signals. In hierarchical designs, these lists may be derived from a single sub-circuit, or from a flattened view of the whole design, depending on the hierarchy mode. For the sake of further discussion, we will assume we are dealing with a single circuit.

If no \$FIND has been done, the \$SIGNALS command will by default operate on all the signals in the circuit, and the \$DEVICES command will operate on all non-pseudo- devices in the circuit. The \$FIND command normally starts by creating a new list of all the objects in the circuit. It then runs down this list and evaluates the first "criterion" item on the line for each object on the list. If the criterion produces any non-null string, the object is kept on the list. This process is repeated by running down the resulting list and evaluating the next criterion for each remaining object.

Once a \$FIND has been done, any objects not on the list are gone as far as subsequent script operations are concerned. Device, signal and pin listings will show only selected objects.

If you have finished with a given selection of devices or signals and want to restore future listings to work on the full set of objects, you can simply do a \$FIND with no criteria:

```
$FIND $DEVICES
```

or

```
$FIND $SIGNALS
```

Using the \$NOCLEAR Option

In some cases, you may want to use some property of a sorted list of objects as a find criterion. An example is finding all cases of two objects having the same value in an attribute field. The way to implement such a search is to sort the list by that value, turn on the merging feature (\$COMBDEVSON/\$COMBSIGSON) and count how many objects get grouped together.

However, this is not normally possible, since \$FIND normally starts by creating a new list derived directly from the design. This destroys any sorting of the existing list. The \$NOCLEAR option solves this by overriding this behavior. Note that this option must appear immediately after \$FIND on the command line.

With \$NOCLEAR specified, \$FIND skips the extraction of a new list and operates on any existing list of objects, in the order that it is currently sorted. This way, you can use the sort as a source of data for the \$FIND. This is illustrated in the following example:

```
$COMBSIGSON
$SORT $SIGNALS $SIGNALNAME
$FIND $NOCLEAR $SIGNALS $GE($COUNT,2)
$CHECK(Duplicate signals found) $SIGNALS
```

In this case we sort the signals by name, then use the \$COUNT function to tell use when we have two or more objects merged together. If \$NOCLEAR was not specified, the sorted order would be lost and \$COUNT would be meaningless.

See also: \$SORT \$COMBDEVSON/\$COMBDEVSOFF \$COMBSIGSON/\$COMBSIGSOFF

Keyword	70 \$FOLDER
Status	
Synopsis	\$FOLDER(<i>string</i>)
Returns	N. A.
Type	Definition
Where	Top level
Description	Used to Specify the default directory for use by design-related file commands, such as \$OPENDESIGN, \$CREATEDIRECTORY and \$CREATEREPORT. The path set by this command will be returned by the \$PROGPATH command.
Example	\$FOLDER(C:\\Program Files\\EMTPWorks\\) \$CREATEREPORT(\$PROGPATH\\myreportfile.txt)

This command does not interpret commands or variable references in the given string. All characters are used verbatim.

A synonym for this command is \$DIRECTORY.

➤ See also: \$PROGPATH

Keyword	71 \$FULLPATH
Status	
Synopsis	\$FULLPATH(<i>string</i>)
Returns	Text
Type	Data
Where	Script
Description	Converts the given directory path name to an absolute path starting with the disk name.
Example	\$FULLPATH(&Dir)

This command is primarily intended to assist in making scripts that are coded only with relative path names for portability while still allowing the generation of full path names for outside processes that require them.

The absolute path name that is generated will use the root directory that contains the current directory as its origin.

See more information about root directories under “File Names and Paths” on page 37.

➤ See also: \$FOLDER \$DESIGNPATH \$SCRIPTPATH \$TEMPPATH \$PROGPATH

Keyword	72 \$GE
Status	
Synopsis	\$GE(string1,string2)
Returns	Boolean
Type	Data
Where	Script
Description	Performs an integer conversion on both arguments and returns TRUE if the first is greater than or equal to the second.
Example	\$IF(\$GE(&ErrCnt, 1))\$ALERT1(&ErrCnt errors!)\$END

This is a numerical comparison, not a string comparison. The comparison is done using signed 32-bit arithmetic.

➤ See also: \$EQ \$GT \$LE \$LT \$NE

Keyword	73 \$GRID
Status	
Synopsis	\$GRID
Returns	Text
Type	Data
Where	Script - any object
Description	Returns a text representation of the location of the object in the drawing grid of the page it is on. For example, an item in the top left corner of a sheet using the default grid layout would return "A1". The results of this command are affected by the setup for the sheet.
Example	\$DEVICES\$DEVNAME &Part \$PAGE-\$GRID This will list devices showing their page and grid location in the schematic.

Keyword	74 \$GT
Status	
Synopsis	\$GT(string1,string2)
Returns	Boolean
Type	Data
Where	Script
Description	Performs an integer conversion on both arguments and returns TRUE if the first is greater than the second.
Example	\$IF(\$GT(&CurDate, &UpdateDate))Time to update!\$END

This is a numerical comparison, not a string comparison. The comparison is done using signed 32-bit arithmetic.

➤ See also: \$EQ \$GE \$LE \$LT \$NE

Keyword	75 \$HEADER
Status	
Synopsis	\$HEADER
Returns	N. A.
Type	Definition
Where	Top level
Description	Defines a block of text to be output whenever a page break occurs.
Example	<pre>\$LINESUSED(30) \$PAGELENGTH(35) \$HEADER Bill of Materials for \$DESIGNNAME, page \$PAGE \$END</pre> <p>The specified text (with appropriate command and variable substitutions) will be output each time the current line count exceeds the \$LINESUSED setting.</p>

➤ See also “Controlling Report Page Layout” on page 20.

Keyword	76 \$HEX
Status	
Synopsis	\$HEX(string)
Returns	Hexadecimal integer
Type	Data
Where	Script
Description	Performs a hexadecimal conversion on the argument. The conversion is done using signed 32-bit arithmetic.
Example	\$DEVICES\$DEVNAME \$HEX(&DateStamp.Dev)

Keyword	77 \$HIERARCHY
Status	
Synopsis	\$HIERARCHY \$FLATDOWN \$FLAT \$CIRCUIT \$TOPCIRCUIT \$PUREDOWN \$PURE
Returns	N. A.
Type	Definition
Where	Top level
Description	Sets the hierarchy mode for the netlist.
Example	<pre>\$HIERARCHY \$FLAT</pre> <p>This specifies that we want to generate a flattened netlist of the entire design, i.e. starting at the topmost circuit level.</p>

The \$HIERARCHY command specifies the hierarchical scope of subsequent \$DEVICES and \$SIGNALS listings and also the type of netlist that will be generated. The scope can be one of:

\$CIRCUIT The current circuit only, i.e. the one in the active schematic window.
This is the default mode if \$HIERARCHY is not specified in a script.

\$TOPCIRCUIT	The topmost circuit in the current design, regardless of which window is active.
\$FLAT/\$PURE	All unrestricted circuit levels in the current design.
\$FLATDOWN/ \$PUREDOWN	All unrestricted circuit levels from the current circuit (the one in the active schematic window) and down.

Netlists that involve more than one circuit level can also be selected to be FLAT or PURE. The options \$CIRCUIT and \$TOPCIRCUIT only deal with a single circuit level and will be flat by definition. Options \$FLAT and \$FLATDOWN generate a "flattened" netlist, that is, one that substitutes the contents of a subcircuit for its parent symbol so that it appears to have been generated from a logically equivalent flat design. Options \$PURE and \$PUREDOWN generate "hierarchical" netlists, i.e. where each internal circuit is defined as a separate netlist and is defined only once.

There are numerous formatting issues in generating netlists from hierarchical designs. These are described in more detail in "Script Hierarchy Issues" on page 32.

Keyword	78 \$HIERNAMESeparator
Status	
Synopsis	\$HIERNAMESeparator(<i>character</i>)
Returns	N. A.
Type	Definition
Where	Top level
Description	Sets the name separator character used in generating \$DEVHIERNAME and \$SIGHIERNAME. The default is a slash "/".
Example	<p>\$HIERNAMESeparator(_)</p> <p>This specifies that we want to generate hierarchical names using an underscore as a separator character. For example "DEV1_GATE1_Q12" or "BLK4_INBUF1".</p>

Keyword	79 \$IF
Status	
Synopsis	<pre> \$IF(conditionString>trueString[\$ELSEfalseString]\$END \$IF(conditionString) trueCommands \$END \$IF(conditionString) trueCommands \$ELSE falseCommands \$END </pre>
Returns	N. A.
Type	Definition
Where	First form – Script Second and third forms - Top level
Description	Provides a mechanism for conditionally including or executing strings. If <i>conditionString</i> is non-null, then <i>trueString</i> is evaluated and included in the output, otherwise <i>falseString</i> (if specified) is used.
Example	<pre>\$IF(&Value),&Value\$END</pre> <p>This example will add a comma and a value to the output line only if the value exists. I.e. This ensures that no comma is output if there is no value.</p>

The \$IF command provides a powerful mechanism to control the format of output and the execution of commands based on text data in the design.

The \$IF command can be used in either of two forms. The single-line form can be used anywhere and requires that the \$IF and its matching \$END (and the \$ELSE, if used) be all on one line. For example:

```
$DEVPINFORMAT $IF($NOT($PINNUM))$SETVAR(Errs, 1)$END
```

The other form of \$IF requires all parts to be on top-level lines, for example:

```

$IF(&Errs)
$NULL($ALERT1(Errors were detected))
$ELSE
$NULL($ALERT1(The circuit is clean!))
$END

```

Keyword	80 \$INCLUDE
Status	
Synopsis	\$INCLUDE \$BLOCK(blockName)[\$RAW] \$INCLUDE[(fileNameString)\$PROMPT[(promptStr)]] [\$RAW] \$INCLUDE[(fileNameString)\$PROMPT[(promptStr)]] \$EXECUTE(blockName)
Returns	N. A.
Type	Definition
Where	Top level
Description	Implements several mechanisms for reading and interpreting text from an external text file or from a block delimited within the script.
Example	\$INCLUDE \$PROMPT(Select the vector file) \$RAW This line will prompt the user to locate a text file which will then be copied verbatim to the output file.

The \$INCLUDE command is used for several different functions, all related to reading text either from an external file or from a block defined within the script.

Including a Named Block

The first form noted above in effect replaces the line containing the \$INCLUDE with the contents of the named block. If the \$RAW option is used, this line has the effect of copying the contents of the block directly to the output file. No processing is done on the contents of the block, other than looking for the \$END line.

If the \$RAW option is not used then the contents of the block are interpreted line by line as if they had been substituted for the \$INCLUDE line itself. This could be thought of as being a sort of subroutine call and is intended to be used similarly. It is useful for cases where the same set of script lines could be repeated at several points.

Including an External File

The second form shown above is similar to the first except that an external text file is read. Again, the \$RAW option determines whether the contents of the file are interpreted as script commands or simply copied to the output.

There are two ways to specify the file to be read. If a name is included in parentheses after \$INCLUDE, then that is taken as the file to read. The file is read from the current directory, which will be the one the current design is located in, unless it has been changed. Note that the file name specification can contain literal text and commands that refer to the current design, such as in the following examples:

```
$INCLUDE($DESIGNNAME.hdr) $RAW
$INCLUDE(&ModelFile) $RAW
$INCLUDE($IF(&HdrFile)&HdrFile$ELSE\Default Header$END)
```

A \$PROMPT option allows you to prompt the user to select a file and optionally specify the text that will appear at the top of the file open box. For example:

```
$INCLUDE $PROMPT(Please select the test vector file) $RAW
```

Using an External File as Data

The third form of \$INCLUDE allows you to read a file line by line and use the contents as data to be interpreted using script commands, rather than as script commands itself. This facility is intended to be used in conjunction with the \$REGEXP regular expression command to implement various types of back annotation or file conversion.

When the \$EXECUTE option is added, the specified block of script commands is executed once for each line in the file. While the block is being executed, the \$TEXTLINE keyword represents the contents of the line. For example, the following script assumes that a file contains a list of device names, one per line. It will display an alert box for every name that is in the file but is not found in the design.

```
$DEFINEBLOCK(FindBlk)
$FIND $DEVICES $EQ($DEVNAME, $TEXTLINE)
$IF($EQ($DEVCOUNT, 0))
$NULL($ALERT1(Missing device $TEXTLINE))
$END
$INCLUDE(Device List) $EXECUTE(FindBlk)
```

To show how you might use a regular expression to extract an old and new name from a file, consider the following simple back annotation:

```
$REPORTOFF
$DEFINEBLOCK(FindBlk)
$IF($NOT($REGEXP((\S+)\s+(\S+), $TEXTLINE)))
$ALERT1(Bad file format at $TEXTLINE)
$ELSE
$FIND $DEVICES $EQ($DEVNAME, &1)
$IF($EQ($DEVCOUNT, 0))
$ALERT1(Missing device &1)
$ELSE
$DEVICES$SETATTR(Name, &2)
$END
$END
$END
$INCLUDE(Device List) $EXECUTE(FindBlk)
```

This simple script assumes that changes are to be applied sequentially, i.e. each line assumes that the previous line's change has been implemented in the design. Some back annotation formats specify all "old names" in the first column and all "new names" in the second. Make sure you know which form is required before using this type of script.

This type of script can make major changes to the data in a design and is not Undoable. It is best to warn the user prominently before proceeding with such drastic changes.

Keyword	81 \$INCLUDEPORTSON/\$INCLUDEPORTSOFF
Status	
Synopsis	\$INCLUDEPORTSON \$INCLUDEPORTSOFF
Returns	N. A.
Type	Definition
Where	Top level
Description	Determines whether Port Connectors are treated as devices for the purposes of netlisting. If ON, port connectors will appear in the list along with all other devices. If OFF (the default), port connectors will not be listed.
Example	\$INCLUDEPORTSON

Port connectors are used in hierarchical designs to link signals in a sub-circuit to a pin on the parent device symbol. Port connector symbols are pseudo-devices and so do not normally appear in any output generated by the Scriptor tool, although their connectivity effect is of course taken into account in generating netlists. For some types of reports, however, it is useful to create a list of a circuit's ports, for example, to generate an interface listing for a circuit. Including the keyword \$INCLUDEPORTSON prior to performing any listing or searching commands will cause port connectors and their pins to be listed as if they were normal devices.

➤ See also: \$ISPORT

Keyword	82 \$INLINE
Status	
Synopsis	\$INLINE(blockName)
Returns	Text
Type	Data
Where	Script
Description	This command is a formatting convenience that allows script commands that normally have to be included on one line to be written out on sequential lines.
Example	<pre> \$DEFINEBLOCK(DevItems) \$IF(\$NOT(\$REGEXP(\w+, \$DEVNAME))) \$IF(\$NOT(\$ALERT2(Name error in \$DEVNAME))) \$ABORT \$END \$END \$END \$DEVICES\$INLINE(DevItems) </pre> <p>This script will behave exactly the same as if the contents of the block were strung together on one line with no line breaks and white space between the items.</p>

This command is a convenience used to overcome the formatting restriction that commands like \$DEVICES or \$SIGPINFORMAT must fit on one line. The contents of the block are interpreted as if they existed in place of the \$INLINE keyword, except that all line breaks and leading and trailing white spaces are ignored.

\$INLINE is intended for use in cases where a \$DEVICES or \$SIGNALS line or a pin format specification requires a complex sequence of commands that will not fit conveniently on one line.

Keyword	83 \$INTERNAL
Status	.
Synopsis	\$INTERNAL[(<i>blockName</i>)]
Returns	Text
Type	Data
Where	Script - device
Description	Causes the internal circuit or the current device (if any) to be output. If no block name is provided, the format for the subcircuit netlist must have been defined in a \$DEFINECIRCUIT block. If a block name is provided, any block defined in a \$DEFINEBLOCK can be used.
Example	<pre> \$HIERARCHY \$PURE \$DEFINEBLOCK(SubCctBlk) \$SORT \$DEVICES \$DEVNAME \$DEVICES\$DEVNAME \$PINS \$END \$DEVICES\$INTERNAL(SubCctBlk) </pre> <p>This will generate a device pin listing for each subcircuit in the design.</p>

See keywords \$DEFINECIRCUIT \$DEFINEBLOCK \$HIERARCHY, and the general description of hierarchical netlists in “Script Hierarchy Issues” on page 32.

Keyword	84 \$ISPORT
Status	
Synopsis	\$ISPORT
Returns	Boolean
Type	Data
Where	Script - device or pin
Description	Returns TRUE ("1") if the current device is a port connector or is the current pin is a pin on a port connector, FALSE (null string) otherwise.
Example	<pre> \$INCLUDEPORTSON \$FIND \$DEVICES \$ISPORT \$DEVICES\PORT \$DEVNAME </pre> <p>This will generate a listing of ports like the following:</p> <pre> PORT IN1 PORT IN2 PORT CLK </pre>

➤ See also \$INCLUDEPORTSON/\$INCLUDEPORTSOFF

Keyword	85 \$ISUNCONNPIN
Status	.
Synopsis	\$ISUNCONNPIN
Returns	Boolean
Type	Data
Where	Script - pin or signal
Description	Returns TRUE ("1") if the current pin is unconnected, that is, it has no visible signal lines or name associated with it on the schematic, FALSE (null string) otherwise.
Example	<pre>\$FIND \$SIGNALS \$ISUNCONNPIN \$SIGNALS\$SIGNALNAME is unconnected!</pre> <p>This will generate a simple error report of unconnected pins.</p>

Keyword	86 \$ITEMSEPARATOR
Status	.
Synopsis	\$ITEMSEPARATOR(<i>string</i>)
Returns	N. A.
Type	Definition
Where	Top level
Description	Sets the string of characters to be inserted between items in a repeating list, i.e. \$PINS or multiple-valued items.
Example	<pre>\$ITEMSEPARATOR(,) \$DEVICES\$DEVNAME \$PINS</pre> <p>This will generate a listing of devices with the pins separated by commas.</p>

➤ See also \$PINS and “Enabling Merging” on page 25.

Keyword	87 \$LE
Status	
Synopsis	\$LE (string1,string2)
Returns	Boolean
Type	Data
Where	Script
Description	Performs an integer conversion on both arguments and returns TRUE if the first is less than or equal to the second.
Example	<pre>\$IF(\$LE(&UpdateDate, &CurDate))Time to update!\$END</pre>

This is a numerical comparison, not a string comparison. The comparison is done using signed 32-bit arithmetic.

➤ See also: \$EQ \$GE \$GT \$LT \$NE

Keyword	88 \$LINESUSED
Status	
Synopsis	\$LINESUSED(<i>number</i>)
Returns	N.A.
Type	Definition
Where	Top level
Description	Sets the number of lines to put on a page before writing out a page header. The actual length of the page is specified by PAGELENGTH above. The default value is 32767, that is, page breaks are disabled.
Example	\$LINESUSED(60)

➤ See also: \$HEADER \$PAGELENGTH

Keyword	89 \$LINEWIDTH
Status	
Synopsis	\$LINEWIDTH(<i>n</i>)
Returns	N. A.
Type	Definition
Where	Top level
Description	Sets the maximum number of characters that will be output on one line by a repeating data item before a line break is inserted.
Examples	\$LINEWIDTH(80) Sets the maximum width of a line to 80 characters.

This command only affects repeating items such as \$PINS or merged data items, so it does not guarantee that no line will be longer than the specified amount. For example, consider the simple script:

```
$LINEWIDTH(20)
$DEVICES$DEVNAME &Part
```

In this case, the \$LINEWIDTH will have no effect because no repeating or multiple-value data item is present. If the total width of the device name, &Part and the separating blank add up to more than 20 characters, they will be simply output with no change.

More information on multiple value items in merged lists is given in “Enabling Merging” on page 25.

Keyword	90 \$LOWERCASE
Status	
Synopsis	\$LOWERCASE(string)
Returns	String
Type	Data
Where	Script
Description	Converts all alphabetic characters in the argument to lower case. Non-alphabetic characters are not changed..
Example	\$LOWERCASE(\$DEVNAME)

See also: \$UPPERCASE

Keyword	91 \$LT
Status	
Synopsis	\$LT(string1,string2)
Returns	Boolean
Type	Data
Where	Script
Description	Performs an integer conversion on both arguments and returns TRUE if the first is less than the second.
Example	\$IF(\$LT(&UpdateDate, &CurDate))Time to update!\$END

This is a numerical comparison, not a string comparison. The comparison is done using signed 32-bit arithmetic.

➤ See also: \$EQ \$GE \$GT \$LE \$NE

Keyword	92 \$MAP
Status	
Synopsis	\$MAP(blockName, string)
Returns	Text
Type	Data
Where	Script
Description	Used to map string values using a predefined table.
Example	<pre> \$DEFINEBLOCK(PkgTable) DIP14 D14-300 DIP16 D16-300 DIP20 D20-300 \$END \$DEVICES\$DEVNAME &Part \$MAP(PkgTable, &Package) </pre> <p>This will output for each device the name, part name and a package code determined by locating the contents of &Package in the first column and outputting the corresponding contents of the second column. Note that the white space shown in this example is representing a single tab character. Blanks do not count as separators and may be included in the values.</p>

This command allows you to map string values by looking them up in a table. This can allow you to translate data in a design that was intended for one usage to a suitable format for a different system. For example, every PCB layout package uses a different set of package codes to select a PCB footprint for a part. The \$MAP command allows you to generate output for a different PCB package than that for which the design was originally created, without making any manual modifications to the codes in the design.

Note that the \$VERIFY command uses the same table format but returns only TRUE ("1") or FALSE (null) to indicate if a given value is present in the table. This can be used for error checking when the actual mapped value is not needed, or may be null.

Mapping Table Format for \$MAP, \$CHARMAP and \$VERIFY

The mapping table used by the \$MAP, \$CHARMAP and \$VERIFY commands is placed in a script *block*, i.e. the first line must be \$DEFINEBLOCK(name) and the last line must be \$END. Note that care must be taken not to include any mapping values that start with a \$ in the first column or contain any tab characters, as this might confuse the script parser. If it is necessary to use such a value, the \$ or tab must be preceded with an escape (backslash) character.

The mapping table consists simply of zero or more lines of text. Each line consists of a match string followed optionally by a tab character and a new value string. The new value string is not used by \$VERIFY and will be ignored if present. The match string must consist of exactly a single character for the \$CHARMAP command. If the new value string is not present, \$MAP or \$CHARMAP will return a null string if it matches the match string. The mapping function searches the strings in the order provided and stops as soon as it finds a match. No checking is done for duplicate values.

Note the following important features of this format:

- The only valid separator character between columns is a tab. Blanks are not considered a separator and may be part of either the match value or the new value string.

- The values provided for both the match string and the new value string are literal text, not script commands. Text that looks like a script command, like "&Package" will be taken as literal text exactly as shown and will not be substituted with the value of the Package attribute. More complex matches can be done using regular expressions and other features by using the \$TABLE command, but execution will be much slower.

➤ See also: \$CHARMAP \$VERIFY

Keyword	93 \$MAXITEMSPERLINE
Status	
Synopsis	\$MAXITEMSPERLINE(<i>numString</i>)
Returns	N. A..
Type	Definition
Where	Script
Description	Specifies the maximum number of items that will be placed on one line by repeating commands such as \$PINS.
Example	<pre>\$DEVPINFORMAT PIN,\$PINNAME,\$PINNUM \$MAXITEMSPERLINE(1) \$DEVICES\DEVICE,\$DEVNAME\$NEWLINE\$PINS</pre> <p>This will generate a listing like the following:</p> <pre>DEVICE,U1 PIN,A,1 PIN,B,2 PIN,X,3 PIN,Ground,7 PIN,Power,14</pre>

This command is used to limit the number of items that will be placed on one line by any repeating command. It is most often used to format pin entries generated by a \$PINS command, but it also affects any item that is repeated due to a merge operation.

➤ See also: Sorting and Merging on page "Sorting and Merging" on page 23 and \$PINS.

Keyword	94 \$MERGE
Status	
Synopsis	\$MERGE
Returns	N.A.
Type	Definition
Where	Script - In \$DEVICES or \$SIGNALS line.
Description	Indicates that all values for objects merged on this line should be output for the next printing item on the line.
Example	<pre>\$SORT \$DEVICES \$TYPENAME \$COMBDEVSON \$DEVICES\$SETATTR(OtherDevs, \$MERGE\$DEVNAME)</pre> <p>In this case, if \$MERGE was not specified, the \$SETATTR command would be iterated at the top level, but the calculation of \$DEVNAME would be done separately for each device. \$MERGE indicates that we want the values of all devices merged in the calculation of the argument to \$SETATTR.</p>

This command is used to override the default behaviour when creating an argument string in a sorted listing. In an example of the type shown above, the device list has been sorted by one field, in this case the type name. Since "\$COMBDEVSON" has been specified, all devices with the same sort value (type name in this case) will be merged onto one line. Normally, any keyword at the "top level" of that line (i.e. not an argument to any other command) will be iterated for all the objects represented by that line. On the other hand, commands that are arguments to other commands will only return the value of the specific device being processed at that point. In other words, the iterating is performed on the top- level results of a command, not on its arguments.

In some cases, it is desired to generate a value based on all the objects merged on a line and pass it to another command. The \$MERGE keyword indicates that the next data keyword should be used as many times as necessary for all objects on the line.

In summary, on the top level of a \$DEVICES or \$SIGNALS line, \$MERGE is the default behavior and it can be overridden by specifying \$SINGLE. When calculating an argument to any other command, \$SINGLE is the default behaviour and it can be overridden by specifying \$MERGE.

➤ See also: \$SINGLE \$SORT \$COMBDEVSON/\$COMBDEVSOFF

Keyword	95 \$MINUS
Status	
Synopsis	\$MINUS(string1,string2)
Returns	Decimal integer
Type	Data
Where	Script
Description	Performs an integer subtraction and returns a decimal string equivalent of the numerical value of the second argument subtracted from the first. The operation is performed with 32-bit signed integers.
Example	\$SETVAR(Count, \$MINUS(&Count, 1))

➤ See also: \$DIV \$MULT \$PLUS

Keyword	96 \$MULT
Status	
Synopsis	\$MULT(string1,string2)
Returns	Decimal integer
Type	Data
Where	Script
Description	Performs an integer multiplication and returns a decimal string of the result.
Example	\$SETVAR(Total, \$MULT(\$COUNT, &PerUnit))

➤ See also: \$DIV \$MINUS \$PLUS

Keyword	97 \$NE
Status	
Synopsis	\$NE(string1,string2)
Returns	Boolean
Type	Data
Where	Script
Description	Performs an integer conversion on both arguments and returns TRUE if they are unequal.
Example	\$IF(\$NE(&COUNT1, 1))Error: Should have exactly one.\$END

This is a numerical comparison, not a string comparison. For example, "1" and "001" will be equal.

➤ See also: \$EQ \$GE \$GT \$LE \$LT

Keyword	98 \$NEWLINE
Status	
Synopsis	\$NEWLINE
Returns	Text
Type	Data
Where	Script
Description	Inserts a line break into the output.
Example	<pre>\$DEVICES\Device \$DEVNAME\$NEWLINE\Type \$TYPENAME</pre> <p>This script will generate two lines per device as in the following example:</p> <pre>Device RLC1 Type RLC</pre>

Keyword	99 \$NEWPAGE
Status	
Synopsis	\$NEWPAGE
Returns	N. A.
Type	Action
Where	Top level
Description	Inserts a page break into the output.
Example	\$NEWPAGE

The \$NEWPAGE command causes these actions to be taken:

If the current page length setting is zero, an ASCII form feed character is written to the output file.

If the current page length setting is greater than zero and the current line count has already exceeded this setting, a single line terminator is written.

If the current page length setting is greater than zero and the current line count is less than this value, then line terminators are written until the page length is reached.

In all cases, if any header has been specified in a \$HEADER section, it is written out.

➤ See also: \$HEADER \$PAGELENGTH and “Controlling Report Page Layout” on page 20.

Although this looks similar to \$NEWLINE, it can only be used on a top-level line and does not return a value that can be used as an argument to another command.

Keyword	100 \$NONBLANK
Status	
Synopsis	\$NONBLANK(<i>string</i>)
Returns	Boolean
Type	Data
Where	Script
Description	Returns TRUE if the given string contains any non-blank characters, FALSE otherwise. Allows you to treat a string of blanks as a "FALSE" value.
Example	<pre>\$DEVPINFORMAT \$IF(\$NOT(\$PINNUM))Error!\$END \$DEVICES\$IF(\$NONBLANK(\$PINS))\$ALERT1(Err in \$DEVNAME)\$END</pre> <p>This script will put up an alert box if any empty pin numbers are found. \$NONBLANK is needed in this case because \$PINS inserts a default blank separator between each pair of pin entries.</p>

Keyword	101 \$NOT
Status	
Synopsis	\$NOT(string)
Returns	Boolean
Type	Data
Where	Script
Description	Performs a Boolean inversion on the argument.
Example	<pre>\$IF(\$NOT(\$ALERT2(OK to continue?)))\$ABORT\$END</pre>

Another, obsolete form of \$NOT is supported for compatibility with 3.x report forms. In a \$FIND line, \$NOT inverts the meaning of the next item on the line, as in:

```
$FIND $DEVICES $NOT $DEVNAME
```

to find unnamed devices. In this version it is preferred to write this as:

```
$FIND $DEVICES $NOT($DEVNAME)
```

for consistency.

Keyword	102 \$NOTES
Status	
Synopsis	\$NOTES text \$END
Returns	N. A.
Type	Definition
Where	Top level
Description	Defines a block of text to be displayed to the script user when the "Format Notes" command is selected. This text has no effect on the script output or action.
Example	\$NOTES The script generates a netlist for Acme PCB. \$END

Notes are not the same as comments (enclosed in "{}"). Notes in a \$NOTES section are specifically intended to give the user information about the usage of the script, required attribute fields, etc.

Keyword	103 \$NULL
Status	
Synopsis	\$NULL(<i>string</i>)
Returns	N. A.
Type	Definition
Where	Script
Description	Discards the value of the argument string.
Example	\$NULL(\$ALERT1(We're still working!))

This command in effect converts any data command into a definition command, eliminating any output from it and eliminating a line terminator that would normally be inserted if it appeared by itself on a line. This is used when it is desired to execute a command for its action, but the value is not wanted in the output.

Keyword	104 \$NUMINPS
Status	
Synopsis	\$NUMINPS
Returns	Decimal integer
Type	Data
Where	Script - device
Description	Returns the number of pins on the device that are defined as INPUT. This is strictly a count of pins defined on the symbol and does not include SIGSOURCE pins and is not summed when multiple symbols are combined onto one line.
Example	\$DEVICES\$DEVNAME \$NUMINPS->\$NUMOUTS

Keyword	105 \$NUMOUTS
Status	
Synopsis	\$NUMOUTS
Returns	Decimal integer
Type	Data
Where	Script - device
Description	Returns the number of pins on the device that are defined as OUTPUT or POWER. This is strictly a count of pins defined on the symbol and does not include SIGSOURCE pins and is not summed when multiple symbols are combined onto one line.
Example	\$DEVICES\$DEVNAME \$NUMINPS->\$NUMOUTS

Keyword	106 \$NUMPINS
Status	
Synopsis	\$NUMPINS
Returns	Decimal integer
Type	Data
Where	Script - device
Description	Returns the number of pins that would be output by a \$PINS command on the current device. This includes all devices merged on the current report line and all SIGSOURCE pins.
Example	\$DEVICES\$DEVNAME \$NUMPINS\$NEWLINE\$PINS

Keyword	107 \$NUMEMTPPINS
Status	
Synopsis	\$NUMEMTPPINS
Returns	Decimal integer
Type	Data
Where	Script – device
Description	Returns the real number of pins on the device. I.e. a 3-phased pin returns 3 EMTP pins.
Example	\$DEVICES\$DEVNAME \$NUMPINS\$NUMEMTPPINS\$PINS

Keyword	108 \$ONEPINSON/ONEPINSOFF
Status	
Synopsis	\$ONEPINSON
Returns	N. A.
Type	Definition
Where	Top level
Description	When ON suppresses the pin number from being printed in netlist entries for devices with only one pin. This will cause entries such as test points to appear as TP1 instead of TP1-1. Default is OFF.
Example	\$ONEPINSON

Keyword	109 \$OR
Status	
Synopsis	\$OR(string1,string2)
Returns	Boolean
Type	Data
Where	Script
Description	Performs a logical OR operation on its two arguments. Any non-null string in an argument is considered TRUE.
Example	\$IF(\$OR(&X1, &X2))&X1/&X2\$ELSE\No data\$END

Keyword	110 \$PAGE
Status	
Synopsis	\$PAGE
Returns	Decimal integer
Type	Data
Where	Script
Description	This command has one of two meanings, depending on context. If used in the context of a device, signal or pin, it refers to the schematic page the object is located on. If it is used in a header, it refers to the page number of the report output.
Example	\$DEVICES\$DEVNAME \$PAGE-\$GRID

➤ See also: \$REPORTPAGE

Keyword	111 \$PAGELENGTH
Status	
Synopsis	\$PAGELENGTH(<i>numLines</i>)
Returns	N. A.
Type	Definition
Where	Top level
Description	This command specifies the total number of lines that make up a page, that is, the number of lines actually used plus the number of blank lines between pages. If this is set to zero, a page break generates a form feed character in the output. The default value is 66.
Example	\$PAGELENGTH(10)

➤ See also \$LINESUSED \$NEWPAGE and “Controlling Report Page Layout” on page 20.

Keyword	112 \$PARENTPIN
Status	
Synopsis	\$PARENTPIN(<i>string</i>)
Returns	String
Type	Data
Where	Script - Pin
Description	Generates a string by evaluating the arguments as if the parent pin was the current pin. Used to extract information about the pin on the parent symbol associated with a pin on a port connector in a subcircuit.
Example	<pre>\$INCLUDEPORTSON \$FIND \$DEVICES \$ISPORT \$DEVPINFORMAT \$IF(\$NOT(\$PARENTPIN(\$PINNAME)))No -match!\$END \$DEVICES\$PINS</pre> <p>This will generate the string "No match" for each pin for which there is no matching parent pin.</p>

This command allows you to get information about the pin on a parent symbol that is associated with a port connector in a hierarchical design. This is intended to be used to generate a port list for a hierarchical Netlist, or for port-to-pin error checking.

The string passed to \$PARENTPIN may contain commands that extract data from the parent pin itself or the parent device itself.

The information that can be extracted using this command is limited to data associated directly with the pin itself or the parent device itself. The attached signal, other pins attached to the same pin, etc. are not available.

If the current pin is not a port or has no associated parent pin, this function will return null.

➤ See also: \$ISPORT \$INCLUDEPORTSON/\$INCLUDEPORTSOFF

Keyword	113 \$PINDIR
Status	

Synopsis	\$PINDIR
Returns	Text
Type	Data
Where	Script – pin
Description	Returns a string representing the direction the pin is facing on the schematic.
Examples	\$DEVPINFORMAT \$PINNAME-\$PINDIR

\$PINDIR returns a two-character code representing the orientation of the current pin, as shown in the following table:

Output String	Direction
E1	East
N1	North
W1	West
S1	South

These codes cannot be modified, although you could make use of the \$MAP function to translate them to another format.

Keyword	114 \$PINNAME
Status	
Synopsis	\$PINNAME
Returns	Text
Type	Data
Where	Script – pin
Description	Returns the pin name associated with the current pin, i.e. as it would appear in the “Edit symbol” option pin list. If the current pin is an automatically-generated SIGSOURCE pin, the name of the associated signal is returned.
Examples	\$DEVPINFORMAT \$PINNAME-\$PINNUM

Keyword	115 \$PINNUM
Status	
Synopsis	\$PINNUM
Returns	Text
Type	Data
Where	Script – pin
Description	Returns the pin number associated with the current pin. If the current pin is an automatically-generated SIGSOURCE pin, the number specified in the SIG-SOURCE attribute is returned.
Examples	\$DEVPINFORMAT \$PINNAME-\$PINNUM

Keyword	116 \$PINS
Status	
Synopsis	\$PINS \$PINS(n) \$PINS(m..) \$PINS(m..n) \$PINS(&attrField)
Returns	Text
Type	Data
Where	Script - device or signal
Description	Returns a list of the pins associated with the current device or signal. The optional argument forms shown above apply only to device pin listings.
Examples	\$DEVICES\$DEVNAME \$PINS

The \$PINS command is used to generate a list of pins associated with a device or signal. Here are some notes on the usage of this command:

- \$PINS will normally appear on either a \$DEVICES or \$SIGNALS line unless the script has been run from a context that implies a current device or signal.
- The rules for sorting the pins before writing them out depend upon whether signal pins or device pins are being listed. These rules are described in the following sections.
- For device pins, the format of each pin is determined by the \$DEVPINFORMAT command, for signal pins, by the \$SIGPINFORMAT command.
- The overall format of the pin list is affected by the \$ITEMSEPARATOR, \$MAXITEMSPERLINE, \$LINEWIDTH, \$CONTSTART, \$CONTEND and other format definition commands.

When used on a \$DEVICES line, \$PINS can take optional arguments that determine the number and order of pins output. The numeric range format is intended to assist in generating netlist formats that require a different position or format for the first pin (typically the output of a logic element) than the remaining pins. This table summarizes the available formats:

\$PINS	All pins are listed.
\$PINS(n)	The single pin whose ordinal position in a complete \$PINS listing would be n is output. If there is no such pin, nothing is output.

\$PINS(m..n)	The pins whose ordinal positions in a complete \$PINS listing would be in the range m..n (inclusive) are output.
\$PINS(m..)	The pins whose ordinal positions in a complete \$PINS listing would be greater than or equal to m are output.
\$PINS(..n)	The pins whose ordinal positions in a complete \$PINS listing would be less than or equal to n are output.
\$PINS(&attrField)	The pins whose names are listed in the given device attribute field are output in the order specified. The pins must be listed in the attribute field by pin name, separated by commas. If the attribute field is empty, no pins will be listed. Any names that don't match are ignored.

Sorting of Pin Entries in Device Pin Listings

Pin entries in a device pin listing are sorted according to the following rules, in decreasing order of priority:

- If an attribute field has been specified in the form \$PINS(&attrField), then exactly the pins listed are output in the order given. Any items that don't match are ignored. If the field is empty, no pins are output.
- If a \$DEVPINSEQUENCE command has been executed and the specified attribute field is non-null, then exactly the pins listed are output in the order given. Any items that don't match are ignored. Unlike the \$PINS(&attrField) format described above, if the attribute field is empty, all pins are output.
- If neither of the above formats apply, then pins are sorted by pin number, whether or not the pin number is displayed.

Sorting of Pin Entries in Signal Pin Listings

The report generator has no explicit method of specifying the sorting to be used for the pins listed within a single signal using the \$PINS statement. The pin list will be sorted by whichever type of device name (\$DEVNAME, \$DEVINSTNAME, \$DEVHIERNAME) is used in the \$SIGPINFORMAT statement. If no name is used, they will be sorted by the Name of the associated device. For pins on the same device, entries will be sorted by pin number.

Keyword	117 \$PINSEQ
Status	
Synopsis	\$PINSEQ[(<i>origin</i>)]
Returns	Decimal integer
Type	Data
Where	Script - pin
Description	Returns a sequence number starting at "n" for the first pin listed on this device or signal. If not specified, zero is assumed for the origin.
Example	\$DEVPINFORMAT \$PINSEQ(1)-\$PINNAME

Keyword	118 \$PINSIGSOURCE
Status	
Synopsis	\$PINSIGSOURCE &fieldName
Returns	N. A.
Type	Definition
Where	Top level
Description	Specifies a pin attribute field from which will be extracted additional pin numbers to be added to the same net as the current pin during \$PINS listings.
Example	\$PINSIGSOURCE &ExtraPins

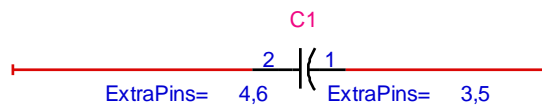
The \$PINSIGSOURCE facility allows multiple pins in a netlist to be represented by a single pin on the schematic. For example, some types of components have multiple package pins joined internally to the same physical point on the device, in order to increase current handling capacity. The \$PINSIGSOURCE command specifies the name of a pin attribute field which will list additional pins to attach to the one on which the attribute is found.

The \$PINSIGSOURCE command line must appear before any \$DEVICES, \$SIGNALS, \$FIND or other commands that may operate on a device or signals list. The field specified must be defined using the Define Attribute Fields command in the Schematic tool. The format of the pin list that is inserted in this field is the same as for the \$SIGSOURCE items, that is, a list of pin numbers separated by commas. If a given pin has no value specified for this field, then its netlist entry will be unaffected.

For example, the following command sequence will add extra pins to the netlist from the field ExtraPins:

```
$PINSIGSOURCE &ExtraPins
$SIGNALS$SIGNAME $PINS
```

The following circuit fragment is shown with the values of ExtraPins shown on each pin. It is important to note that this is an attribute of each individual pin, not of the associated device or signal.



Pins 1 and 2 are specified using the normal pin numbering scheme. The items in ExtraPins will be added to the same net as the associated pin. In a typical Netlist format, this will generate the following results:

```
SIG1 C1-2 C1-4 C1-6
SIG2 C1-1 C1-3 C1-5
```

The \$PINSIGSOURCE command will significantly slow down report generation because every pin listed in the Netlist must be searched for the specified attribute field. For this reason, we do not recommend inserting it as a general-purpose addition to your Netlist forms if you are not using it.

➤ See also: \$DESIGNPINSIGSOURCE/\$DESIGNSIGSOURCE \$SIGSOURCE

Keyword	119 \$PINTYPE
Status	
Synopsis	\$PINTYPE
Returns	String
Type	Data
Where	Script - pin
Description	Returns a string representing the type of the current pin. The string used for each pin type can be set using the \$PINTYPEFORMAT command.
Example	\$DEVPINFORMAT \$PINNAME\(\$PINTYPE\)

Keyword	120 \$PINTYPEFORMAT
Status	
Synopsis	\$PINTYPEFORMAT literalString1 literalString2 ...
Returns	N.A.
Type	Definition
Where	Top level
Description	Specifies the string used to describe each pin type, i.e. the string that will be returned by \$PINTYPE.
Example	<p>\$PINTYPEFORMAT I O O O O O O O O O O O O O O O P</p> <p>This indicates that the single letter I will be returned by \$PINTYPE for input pins, O for output and P for power pins.</p>

This format command specifies the way the type (i.e. input, output, etc.) of a device pin is presented in reports. Whenever the \$PINTYPE keyword is encountered, the program uses the pin type of the current pin to look up a string in this list and that is substituted into the output.

The \$PINTYPEFORMAT command must be followed on the line by a sequence of literal strings separated by blanks which described the appearance of each item. The strings are given in the following order, separated by blanks:

\$PINTYPEFORMAT Item Order

Item order	Function	Default string
1	Input	INPUT
2	Output	OUTPUT
3	3-state	3STATE
4	Bidirectional	BIDIR
5	Not used	?????
6	Open Collector	OC
7	Undefined	?????
8	Bus	BUS
9	Perm. low	LOW
10	Perm. high	HIGH
11	Latched input	LTCHIN
12	Latched output	LTCHOUT

13	Clocked input	CLKIN
14	Clocked output	CLKOUT
15	Clock input	CLOCK
16	Open Emitter	OE
17	No connection	NC
18	Power	POWER

“Undefined” can only occur if a device was being created in the DevEditor and was closed before all its pins had been specified.

If less than 18 strings are specified, the trailing items retain their default value.

Note however that there are presently only three types of pins available with EMTPWorks circuits: INPUT (1), OUTPUT(2) and POWER(18).

Generating a Warning for Invalid Pin Types

The \$PINTYPEFORMAT command also provides the ability to generate a warning to the user if a pin type is used that is invalid for the type of output being generated. If the keyword \$WARN is inserted in the list of type strings, if any of the following items are used, a warning will be generated. \$NOWARN turns this back off again. For example:

```
$PINTYPEFORMAT I O O B $WARN ? ? ? ? ? ? ? ? ? ? ? ? ?
```

This specifies that if any pin type other than the ones specifically allowed are found in the design, the user will be warned and a "?" will be placed in the output.

➤ See also: \$PINTYPE

Keyword	121 \$PLUS
Status	
Synopsis	\$PLUS (string1,string2)
Returns	Decimal integer
Type	Data
Where	Script
Description	Performs an integer addition and returns a decimal string of the result. All arithmetic is done with 32-bit signed values.
Example	\$SETVAR(ErrCount, \$PLUS(&ErrCount, &NewErrs))

➤ See also: \$DIV \$MINUS \$MULT \$PLUS

Keyword	122 \$PORTNAME
Status	
Synopsis	\$PORTNAME
Returns	Text
Type	Data
Where	Script - Pin
Description	If the current pin is a port, this returns its port name, if not, null. This is used to get the name of a port connector from within a subcircuit. This is only valid for port pins, i.e. when \$ISPORT is TRUE.
Example	<pre>\$INCLUDEPORTSON \$FIND \$DEVICES \$ISPORT \$DEVPINFORMAT \$PORTNAME \$DEVICES\$PINS</pre> <p>This will produce a list of all the ports in a circuit.</p>

This keyword is used to create a port list when working from a subcircuit upwards. It returns the "port name" associated with a pin on a port connector, i.e. the name that will be used to associate the port with a parent symbol. For any non-bus port connector, there is only one pin and this will return the name applied by the user to the port connector symbol itself. For bus port connectors, for pin 1 (i.e. the bus pin) it will return the port connector name. For pins 2 and up (i.e. the internal signal pins), it will return the pin name.

The keyword is not valid for non-port connector pins.

➤ See also: \$ISPORT \$INCLUDEPORTSON/\$INCLUDEPORTSOFF

Keyword	123 \$PRIMNAME
Status	
Synopsis	\$PRIMNAME
Returns	String
Type	Data
Where	Script - device
Description	Returns the primitive type name of the current device, that is, the name of the EMTWorks primitive type that was used in creating the symbol.
Example	\$DEVICES\$DEVNAME \$TYPENAME(\$PRIMNAME\)

Keyword	124 \$PROGPATH
Status	
Synopsis	\$PROGPATH
Returns	Text
Type	Data
Where	Script – design
Description	Returns the path set by \$FOLDER or \$DIRECTORY commands.
Examples	My default folder is: \$PROGPATH

➤ See also: \$FOLDER

Keyword	125 \$PROGRESS
Status	
Synopsis	\$PROGRESS(<i>string</i>)[\$ON \$OFF][\$PERCENTON \$PERCENTOFF] \$PROGRESS(<i>string</i>)
Returns	N.A.
Type	Definition
Where	First form - Top level. Second form – Script
Description	Controls the display of the progress window.
Example	\$DEVICES\$PROGRESS(Now generating \$TYPENAME)\$INTERNAL This will display the current type name in the progress window.

This command allows you to override the default behaviour of the progress window that is displayed during script execution. Especially in complex scripts, the display of percentages may be meaningless and it may be better to display a single message indicating the current function. In other case, you may want the script to execute invisibly by disabling the progress window completely.

When it appears at the top level of a script (i.e. on a line by itself) \$PROGRESS can take a number of options:

- \$PERCENTOFF - This disables the display of the percentage complete.
- \$PERCENTON - This enables the display of the percentage complete for subsequent iterating commands, e.g. \$DEVICES.
- \$OFF - This completely disables the progress window, removing it from the screen.
- \$ON - This re-enables the progress window or has no effect if it is already displayed.

When it appears in a script line such as \$DEVICES, \$PROGRESS can only be used to specify the string that will appear in the window.

Keyword	\$PROMPT1/\$PROMPT2
Status	
Synopsis	\$PROMPT1(<i>message</i>)
Returns	String
Type	Data
Where	Script
Description	Displays a prompt box to the user with the given message text. \$PROMPT1 displays only one button, OK. \$PROMPT2 displays two buttons, OK and Cancel. Returns the entered string if OK, NULL if Cancel.
Example	\$PROMPT1(Enter your name please)

➤ See also: \$ALERT1/\$ALERT2

Keyword	126 \$REGEXP
Status	
Synopsis	\$REGEXP(regularExpression,string)
Returns	Boolean
Type	Data
Where	Script
Description	Matches the regular expression supplied against the given string. May also set the values of match variables &1 through &9. Returns TRUE if the match is successful, FALSE otherwise.
Example	<pre>\$IF(\$REGEXP.(*(\d+).*, &Value))The numeric part is &1\$END</pre> <p>The given regular expression means "match any sequence of characters followed by one or more digits followed by any sequence of characters". Because the "digits" part of the expression is in parentheses, it will be assigned to match variable &1 if it is successful. If \$REGEXP is successful, it returns TRUE ("1") and the numeric value will be output.</p>

This command applies a Unix-style regular expression against a given string. This can be used to check the format of any text values in a design or to extract sub-fields out of any data string, as shown in the above example. This can be used in conjunction with the \$INCLUDE \$EXECUTE command to read arbitrary text data from a file for back annotation or other purposes.

Because regular expressions are a major topic, they are covered in more detail in "Regular Expressions" on page 39.

Keyword	127 \$REPLICATE
Status	
Synopsis	\$REPLICATE(string,char1, ,char2)
Returns	Text
Type	Definition
Where	Top level
Description	Replaces <i>char1</i> by <i>char2</i> in <i>string</i> and returns the result.
Example	<pre>\$SETVAR(_Test,It is a test) \$REPLICATE(&_Test,a, ,b)</pre> <p>The output will be: It is b test</p>

Keyword	128 \$REPORTON/\$REPORTOFF
Status	
Synopsis	\$REPORTON \$REPORTOFF
Returns	N.A.
Type	Definition
Where	Top level
Description	Enables and disables report output.
Example	\$REPORTOFF

These commands are used to enable or disable the generation of an output file during script execution. By default, any literal data in a script or any data generated by commands is output to a text file. If no output file exists at the time this data is encountered, the user is prompted to create a file. This behaviour may not be desired in cases where a script is being used for non-reporting purposes such as updating values in a design. In this case, the \$REPORTOFF command disables the default file output. Note that transcript output is not affected.

➤ See also: \$CREATEREPORT

Keyword	129 \$REPORTPAGE
Status	
Synopsis	\$REPORTPAGE
Returns	Decimal integer
Type	Data
Where	Script
Description	This command returns the page number of the report output.
Example	\$HEADER Page Number: \$REPORTPAGE Date: \$DATE \$ENDHEADER

➤ See also: \$PAGE

Keyword	130 \$SAMEPINCOUNT
Status	
Synopsis	\$SAMEPINCOUNT
Returns	Decimal integer
Type	Data
Where	Script - Pin
Description	Returns a decimal integer indicating how many pins on the same device as the current pin have the same pin number. Used to detect duplicate pin numbers on devices.
Examples	<pre>\$SORT \$DEVICES \$DEVNAME \$COMBDEVSON \$DEVPINFORMAT \$IF(\$GT(\$SAMEPINCOUNT, 1))Duplicate pin!\$END \$DEVICES\$IF(\$NONBLANK(\$PINS))\$SELECT\$END</pre> <p>This script will select any device in the circuit with duplicate pin numbers. The sort is necessary to merge symbols with the same package name.</p>

This keyword is valid in a device pin format and is used to detect cases of duplicate pin numbers on a device. Note that if two pins have the same pin number and are attached to the same signal, they are not counted as duplicates. This is done because power and ground pins in attributes and other common pins are frequently repeated on multiple symbols assigned to the same package. As long as there is no conflict, this is not considered to be an error.

Keyword	131 \$SCRIPTPATH
Status	
Synopsis	\$SCRIPTPATH
Returns	Text
Type	Data
Where	Script – design
Description	Returns the script file path
Examples	My script directory is: \$SCRIPTPATH

Keyword	132 \$SELECT
Status	
Synopsis	\$SELECT
Returns	N.A.
Type	Definition
Where	Script - any object
Description	Selects the current object on the schematic, i.e. highlights it for user operations.
Examples	<pre>\$SORT \$DEVICES \$DEVNAME \$COMBDEVSON \$DEVPINFORMAT \$IF(\$GT(\$SAMEPINCOUNT, 1))Duplicate pin!\$END \$DEVICES\$IF(\$NONBLANK(\$PINS))\$SELECT\$END</pre> <p>This script will select any device in the circuit with duplicate pin numbers.</p>

This keyword is used to create interaction between the execution of the script and the user. It allows the script to select objects in the schematic so that they can be located by the user with the "Go To Selection" command or operated on with the Browser tool, etc.

Keyword	133 \$SELECTED
Status	
Synopsis	\$SELECTED
Returns	Boolean
Type	Data
Where	Script - any object
Description	Returns TRUE (1) if the current object is in a selected state on the schematic. Allows the script to locate objects selected by the user on the schematic for subsequent operations.
Examples	<pre>\$FIND \$DEVICES \$SELECTED \$DEVICES\$DEVNAME</pre> <p>Makes a list of the selected devices.</p>

Keyword	134 \$SETATTR
Status	
Synopsis	\$SETATTR(fieldName, string)
Returns	N.A.
Type	Definition
Where	Script - any object
Description	Sets the given attribute field to the given string value in the current object. If the field does not exist, it is created.
Examples	\$SETATTR(ReportDate, \$DATE(\$R))

The \$SETATTR command allows you to define and set attribute fields to a specified string value. This command can be used anywhere, e.g. on a top-level line of the script, on a \$DEVICES or \$SIGNALS line or within a \$SIGPINFORMAT or \$DEVPINFORMAT. This command always operates on the "current object" which may depend on context. In particular, if used in a \$SIGPINFORMAT or \$DEVPINFORMAT, it always operates on the current pin. If you want to

extract information from a pin and apply it to the parent device or signal, you have to assign it to a variable and apply it to the device or signal on the \$DEVICES or \$SIGNALS line.

See more information about the current object in “Current Design or Current Object” on page 14.

If the given attribute field is not defined in the design’s attribute table, it will be added automatically with default settings. It generally preferable to define a field explicitly using \$DEFINEATTR before setting it to ensure that you get the definition settings you want, e.g. Primary vs. Secondary.

Keyword	135 \$SETERRORBIT
Status	
Synopsis	\$SETERRORBIT(<i>bitNum</i>)
Returns	N. A.
Type	Definition
Where	Script - Any object
Description	Sets the given bit number in the binary error set represented in the object’s “OKErrors” attribute field.
Examples	\$FIND \$DEVICES \$SELECTED \$DEVICES\$SETERRORBIT(7) Sets the "Mark as OK" setting on the given bit on each device that is selected in the circuit.

This call is one of a set of commands designed to implement a "Mark as OK" feature in error checking scripts.

- See also: \$CLEARERRORBIT \$CLEARERRORS \$ERRORBITON \$ERRORBITOFF and “Implementing Mark as OK in Error Checking Scripts” on page 28.

Keyword	136 \$SETSIGWIDTH
Status	
Synopsis	<code>\$SETSIGWIDTH(<i>numString</i>)</code>
Returns	N.A.
Type	Definition
Where	Script - Signal
Description	Sets the given "line width" setting for the given signal to the integer value expressed in decimal in <i>numString</i> .
Examples	<pre>\$FIND \$SIGNALS \$SELECTED \$SIGNALS\$SETSIGWIDTH(3)</pre> <p>This script will set the width on selected signals in the current circuit to 3.</p>

This command intended to assist in applications where signal lines are used to represent vectors or busses and it is desired to show this graphically using different line widths on the schematic. The normal width for a signal line is 1.

Keyword	137 \$SETVAR
Status	
Synopsis	<code>\$SETVAR(varName, string)</code>
Returns	N.A.
Type	Definition
Where	Script
Description	Sets the given variable to the given string value. If the variable does not exist, it is created.
Examples	<code>\$SETVAR(ErrMsg, The number of errors was &NumErrs)</code>

The \$SETVAR command allows you to define and set variables to a specified string value. This command can be used anywhere, e.g. on a top-level line of the script, on a \$DEVICES or \$SIGNALS line or within a \$SIGPINFORMAT or \$DEVPINFORMAT. Variables are "global" and "static" in the sense that a variable set anywhere will hold its value and be known anywhere else in the script until it is explicitly changed. For example, you can set use \$SETVAR to set a variable to some value in a \$SIGPINFORMAT used by a \$SIGNALS \$PINS listing. The last value set will still be known after the \$SIGNALS listing is complete.

Note that an empty string is a valid string value, so \$SETVAR(BadPins,) is a valid usage of this command.

IMPORTANT NOTE REGARDING VARIABLE REFERENCES

If a variable is not known at the time a reference to it is encountered by the script parser, a null value is assumed. This is done for two reasons:

The parser cannot distinguish between attribute references and variable references because they both use the same syntax. It therefore doesn't want to create a variable in response to this reference if it was intended as an attribute reference.

For efficiency, references to variables and attribute fields that don't exist are compiled out so that repeated executions will not waste time looking the values up.

Because script lines are compiled a line at a time, unexpected results may occur if you refer to a variable value before the \$SETVAR that sets it on the same line. For example:

```
$DEVICES$IF($NOT(&BeenHere))First Time!$END$SETVAR(BeenHere, X)
```

When the reference to BeenHere is encountered while compiling this line, the variable does not exist and so it is ignored, i.e. assumed to be null. Even though the value is set later on the line, the text "First Time" will be output for every device in this listing.

The solution to this problem is to always do a \$SETVAR on a line by itself to define the variable if there is any chance that it may be referred to before the first real value set. In this case:

```
$SETVAR(Beenhere, )  
$DEVICES$IF($NOT(&BeenHere))First Time!$END$SETVAR(BeenHere, X)
```

will produce the desired results.

Keyword	138 \$SIGCOUNT
Status	
Synopsis	\$SIGCOUNT
Returns	Decimal integer.
Type	Data
Where	Script
Description	The number of items on current signal list. If COMBSIGSON has been set, this will mean in effect the number of different sort values in the list.
Examples	\$SIGNALS \$ALERT1(the number of items on current signal list is: \$SIGCOUNT)

\$SIGCOUNT is valid only after a \$SORT \$SIGNALS, a \$FIND \$SIGNALS, or a \$SIGNALS line has been executed.

Keyword	139 \$SIGHIERNAME
Status	
Synopsis	\$SIGHIERNAME
Returns	Text
Type	Data
Where	Script - signal
Description	Returns the hierarchical name of the current signal, i.e. the signal name prefixed with the names of any parent devices in the hierarchy. In Flat designs, this is equivalent to \$SIGNALNAME. This is not valid in Pure designs since there is no unequivocal path from a given signal to the top-level circuit.
Examples	<p>\$SIGNALS\$SIGHIERNAME \$PINS</p> <p>This will generate a list of signals showing the hierarchical name of each followed by a list of the attached pins.</p>

\$SIGHIERNAME is intended to provide a unique signal identifier when generating flattened netlists from hierarchical designs. Here are some important points to remember when using this keyword.

The hierarchical name consists of the Name value for the current signal with the "path" formed by its parent devices prefixed to it, such as:
MEMBLK1/CONTROL/CTR2

The separator character that is used in generating these names can be set using the \$HIERNAMESEPARATOR command.

Hierarchical names can be arbitrarily long, depending on the nesting level. For this reason, they may not be suitable as an identifier in many Netlist formats.

➤ See also \$HIERNAMESEPARATOR and "Script Hierarchy Issues" on page 32.

Keyword	140 \$SIGINSTNAME
Status	
Synopsis	\$SIGINSTNAME
Returns	Text
Type	Data
Where	Script - signal
Description	Returns the instance name of the current signal, i.e. the contents of its InstName attribute field.
Examples	<p>\$SIGNALS\$SIGINSTNAME \$PINS</p> <p>This will generate a list of signals showing the instance name of each followed by a list of the attached pins.</p>

The signal instance name is not automatically assigned by the program. \$SIGINSTNAME is not normally used to identify signals in Netlists for this reason. It is more common to use \$SIGHIERNAME unless the target system has a strict name length or character set limit that makes this impossible. If \$SIGINSTNAME is used it is the user's responsibility to ensure that every signal in the design has been assigned a unique value.

Keyword	141 \$SIGLOC
Status	
Synopsis	\$SIGLOC
Returns	Text
Type	Data
Where	Script - Signal
Description	Will be replaced the "locator" or hierarchical token number of the current signal object.
Examples	\$SIGNALS\$SIGHIERNAME \$SIGLOC

The \$SIGLOC command will be replaced by a unique identifier string referred to as a "locator". This can be used to uniquely identify any signal within a hierarchical design. The locator is used by other EMTPWorks modules like ErrorFind to locate objects unambiguously without relying on internal memory addresses.

The format of the locator string is described elsewhere in this manual.

Keyword	142 \$SIGNALS
Status	
Synopsis	\$SIGNALS formatItems
Returns	Text
Type	Data
Where	Top level
Description	Creates a listing of signals using the format items following on the line.
Examples	<p>\$SIGNALS\$DEVNAME \$PINS</p> <p>This will generate a list of signals showing the name of each followed by a list of the attached pins.</p>

The \$SIGNALS keyword also appears as a modifier of the \$FIND, \$SORT, \$ASSIGNNAMES, \$ASSIGNINSTNAMES and \$BREAK commands. See the entries for those commands for more information.

The \$SIGNALS command is the primary command for generating any listing by signal. The format of the listing is determined by the format items that follow the command on the line and by the various definition options, as outlined below.

The scope of the listing in hierarchical designs is affected by the \$HIERARCHY command.

A subset of the signals in the design can be extracted using any desired criteria using the \$FIND command.

The ordering of signals in the listing is affected by the \$SORT command.

If the \$COMBSIGSON has been selected, multiple items with the same sort value will appear on one line.

The \$NEWLINE format command can force multiple lines to be generated for each item.

Limits on line length imposed by the \$LINEWIDTH and \$MAXITEMSPERLINE commands may cause multiple lines to be generated.

he format of the listing for each signal is completely determined by the text and commands which follow the \$SIGNALS keyword. There is no default format, so if no line format is specified, a sequence of empty lines will be written to the file.

Any characters appearing after \$SIGNALS that is not part of a format sub-command will be placed verbatim in each line of the listing. For example, blanks, tabs or commas can be used to format each line.

Numerous data-generating commands can be used in defining the output generated on each line. Any command listed with a "Where" value of "Script - Signal", "Script - Circuit", "Script - Any object" or "Script" can be used on a \$SIGNALS line. Some of the more common format items are summarized in this table. See the corresponding keyword section for each item for more information.

\$SIGNAL	The name of the signal, that is, the contents of its Name attribute field.
\$SIGSEQ(n)	The sequence number of the current signal, i.e. an integer assigned to the signal, starting at "n" for the first device in the sorted sequence and incremented for each subsequent one.
\$COUNT	The number of signals being merged to form this line, i.e. the number having the same sort value.
\$SINGLE	Tells the report generator to write only a single value for all items following on the line, regardless of how many items were combined with the same sort value.
\$NUMPINS	The number of pins attached to the signal.
\$PAGE	The circuit page that this signal appears on. Since signals can connect across pages, this can generate multiple values.
\$PINS	A list of the signal's pins. The format of each item is determined by the \$SIGPINFORMAT command.
\$COL(N)	If the current position in a line is less than N spaces from the left hand end then blanks are inserted until the Nth column is reached.
\$NEWLINE	Writes a new line character into the file. This allows a device entry to occupy several lines in the output report.
&attr	Inserts the value of the attribute field specified.

A variety of applications for the \$SIGNALS command are described in "Script Examples" on page 44.

Keyword	143 \$SIGNAME
Status	
Synopsis	\$SIGNAME
Returns	Text
Type	Data
Where	Script – signal
Description	Returns the name of the current signal, i.e. the contents of its Name attribute field. The name can be optionally prefixed with the name of any enclosing bus by using the \$BUSNAMEON command.
Examples	<p>\$SIGNALS\$SIGNAME \$PINS</p> <p>This will generate a list of signals showing the name of each followed by a list of the attached pins.</p>

➤ See also: \$BUSNAME \$BUSNAMEON/\$BUSNAMEOFF

Keyword	144 \$SIGPINFORMAT
Status	
Synopsis	\$SIGPINFORMAT formatItems \$SIGPINFORMAT(formatItems)
Returns	Text
Type	Data
Where	First form - Top level Second form - Script – signal
Description	Sets the format that will be used for each pin output by the next \$PINS command in a \$SIGNALS line. The first form (without parentheses) can only be used on a top-level line. The second form (with parentheses) can be used either on a top-level line or in a \$SIGNALS line. This allows the format to be modified on the fly while generating output for a single signal.
Examples	<pre>\$SIGPINFORMAT [PINNAME,PINNUM] \$ITEMSEPARATOR(,) \$SIGNALS\$SIGNAME \$PINS</pre> <p>This will generate a list of signals showing the name of each followed by a list of the attached pins using the format specified:</p> <pre>U1 [A,1],[B,2],[Q,3]</pre>

See the notes on the usage of pin formats under “\$DEVPINFORMAT\$DEVPINFORMAT” command.87

Keyword	145 \$SIGSEQ
Status	
Synopsis	\$SIGSEQ[(<i>origin</i>)]
Returns	Decimal integer
Type	Data
Where	Script – signal
Description	Returns an integer indicating the position in the sort sequence of this signal. If the original is not specified, numbering starts at zero.
Examples	<pre>\$SORT \$SIGNALS \$SIGNAME \$SIGNALS\$SIGSEQ(1). \$SIGNAME</pre> <p>This will generate a listing of devices that looks something the following:</p> <pre>1. C1 2. C2 3. U1 4. U2</pre>

\$SIGSEQ returns the sequence number of the current device, that is, an integer assigned to the device, starting at "origin" for the first device in the sorted sequence and incremented for each subsequent one. This number is not permanently associated with the device but it strictly its sequence number in the current sort. The "(origin)" part is optional. If it is omitted the sequence starts at zero. If no sort has been done, all devices will have the origin sequence number.

Keyword	146 \$SIGSOURCE
Status	
Synopsis	\$SIGSOURCE(sigName) [&fieldName]
Returns	N. A.
Type	Definition
Where	Top level
Description	Defines a "signal source" attribute field. If <i>fieldName</i> is not specified, it is assumed to be the same as the signal name.
Examples	<p>\$SIGSOURCE(Plus5V) &Power</p> <p>This will cause the attribute field "Power" to be searched in each device for pin numbers to add to the net "Plus5V".</p>

The "signal source" facility allows you to name certain signals to be treated as power and ground nets. This has the following two effects:

- It allows you to create common connections in a circuit using attribute entries in devices. This can be used to create power and ground entries in the netlist without having to show all these connections explicitly on the diagram.
- It informs the Netlist generator that the given signals should be merged across hierarchy levels.

See more information on power and ground nets in hierarchy in "Script Hierarchy Issues" on page 32.

Any device attribute field may be specified as a signal source by the command:

```
$SIGSOURCE(signalName) &fieldName
```

signalName is the name of the signal to attach pins to in the netlist. *fieldName* is the name of the attribute field to search for in each device to look for pin numbers to attach to the named signal. If &fieldName is omitted, the signal name is taken to be the name of the attribute field to search.

The named device attribute field is assumed to contain a list of pin numbers to attach to the signal. This can consist of a single pin entry, such as "7", or a list, such as "5,6,9,14".

For example, Ground connections can be created as follows:

- Using the Attributes command on a selected device, create an entry such as "7" in the Ground attribute for a device, in this case connecting pin 7 to the Ground net.
- Place the command \$SIGSOURCE(Ground) in the script. This causes Report to search all device attributes for fields named Ground and use the field value as a pin number.

There is no fixed limit to the number of SIGSOURCE entries that can be created.

The predefined fields Ground and Power are normally used for standard power and ground connections. These pin connections are prespecified for all digital components in the standard EMTWorks libraries. Corresponding \$SIGSOURCE statements are included, where appropriate, in all standard netlist script files. You can create your own special-purpose power nets by using the Define Attribute Fields command to define a new field, then adding the

appropriate SIGSOURCE statement to the netlist script. Alternatively, the \$DESIGNSIGSOURCE command can be used to allow special-purpose signal sources to be specified per design.

- See also: \$DESIGNPINSIGSOURCE/\$DESIGNSIGSOURCE and “Reporting Power and Ground Nets” on page 30.

Keyword	147 \$SIGTOKEN
Status	
Synopsis	\$SIGTOKEN
Returns	Decimal integer
Type	Data
Where	Script - signal
Description	Returns an integer representing the signal's token number.
Examples	<div>\$SIGNALS\$SIGNAME \$SIGTOKEN</div> <div>This will generate a listing of signals with the name and token number of each.</div>

Keyword	148 \$SINGLE
Status	
Synopsis	\$SINGLE
Returns	N.A.
Type	Definition
Where	Script - In \$DEVICES or \$SIGNALS line.
Description	Indicates that only a single object value should be output for the next printing item on the line.
Example	<pre>\$SORT \$DEVICES \$TYPENAME \$COMBDEVSON \$DEVICES\An example of a \$TYPENAME is \$SINGLE\$DEVNAME</pre> <p>In this case, if \$SINGLE was not specified, \$DEVNAME would cause the names of all devices having the same type name to be output on that line. \$SINGLE indicates that we only want one.</p>

The \$SINGLE command is used to override the default "merging" behaviour when outputting a sorted listing. In an example of the type shown above, the device list has been sorted by one field, in this case the type name. Since "\$COMBDEVSON" has been specified, all devices with the same sort value (type name in this case) will be merged onto one line. If any keyword appears on that line that generates data derived from a device, all the values associated with objects included in that line will be shown. In this case, the \$DEVNAME keyword would cause a list of the names of all devices of each type to appear together on the line.

In some cases, it is desired to put out only a single value on a line, regardless of how many objects are represented. The \$SINGLE keyword indicates that the next data keyword should be used only once. Note that only the first item found on the list is output and there is no guarantee of which object this will be or that the same order will be preserved after the design has been edited.

➤ See also: \$MERGE \$COMBDEVSON/\$COMBDEVSOFF \$SORT

Keyword	149 \$SORT
Status	
Synopsis	\$SORT \$DEVICES \$SIGNALS [\$ASCENDING \$DESCENDING] [\$LEXICAL \$NUMERIC] [\$NULLLOW \$NULLHIGH] sortItems
Returns	N. A.
Type	Action
Where	Top level
Description	Sorts the device or signal list according to the sort items specified.
Examples	<pre>\$SORT \$DEVICES \$TYPENAME</pre> <p>This will generate a listing of devices sorted by their type name.</p>

The \$SORT command provides the ability to sort device and signal listings on any data field. Items with the same value in any field can be optionally merged into a single line. This allows listings to be organized to suit various applications, for example:

- Device or signal lists can be sorted by name to enhance readability.

- Device lists can be sorted by type so that each line lists one device type with all information common to that type and a list of instances of the type.
- A device list can be sorted by page number to show the devices used on each page.
- Signals can be sorted by an attribute field, for example to give priority to certain nets so they are listed first for autorouting purposes.
- Devices can be sorted by any attribute field, for example by component value, stock number, cost, etc.

The \$SORT command has the following basic form:

```
$SORT objectType field1 field2 ...
```

objectType must be either "\$DEVICES" or "\$SIGNALS".

field1, *field2*, etc. are identifiers indicating which fields to sort on. The list is sorted first on the first field. If any items in the list have identical values in that field, then the groups of like-valued items are sorted on the next specified field, etc.

Sort fields are limited to the specific keywords shown in the table below. In this version, it is not possible to use arbitrary Scripter functions and expressions as sort values.

Once a sort has been done, it remains in effect for all subsequent listings on that object type until the next \$SORT or \$FIND command. Each \$SORT clears the previous \$SORT.

Sort Options

A number of options can be specified that will affect the sorted order:

\$ASCENDING/ \$DESCENDING	Specifying one of these two keywords indicates the desired sort order. If \$ASCENDING is given, the list will be sorted by increasing values of the sort key, \$DESCENDING will produce the opposite order. \$ASCENDING is the default.
\$LEXICAL/ \$NUMERIC	These options determine how numeric values are treated as part of a sort value. If \$LEXICAL is specified, sorting is done strictly on ASCII character values. If \$NUMERIC is specified, any sequence of numeric characters found in a sort item is taken as an integer value. For example, if you were sorting three devices named U2, U9 and U11, a lexical sort will produce U11, U2, U9, whereas a numeric sort will produce U2, U9, U11. The numeric sort is generally preferable for items numbered with integers, whereas the lexical sort is better for component values, part numbers or other cases where a value should just be treated as a sequence of characters even if it contains digits. \$NUMERIC is the default.
\$NULLLOW/ \$NULLHIGH	These options determine the handling of null valued items, i.e. empty strings. If \$NULLLOW is set, null items will be considered to have a low value. In other words, if an \$ASCENDING sort is done, they will appear first. \$NULLHIGH is the default.

Sort Field Keywords

The following table describes the sort fields available.

\$\$SORT Item Keywords

Sort Item Keyword	Object Type	Meaning
\$DEVNAME	\$DEVICES	The device name.
\$DEVHIERNAME	\$DEVICES	The device hierarchical name.
\$DEVINSTNAME	\$DEVICES	The device instance name.
\$SIGNALNAME	\$SIGNALS	The signal name.
\$SIGHIERNAME	\$SIGNALS	The signal hierarchical name.
\$SIGINSTNAME	\$SIGNALS	The signal instance name.
\$SIGSOURCE	\$SIGNALS	An integer value that is 1 for the first SIGSOURCE name declared in the script, 2 for the next one, etc. Nets that are not declared as SIGSOURCES get a high value. This is used to give a higher priority to power and ground nets in netlists.
\$POSX	\$DEVICES	The X position on the schematic.
\$POSY	\$DEVICES	The Y position on the schematic.
\$TYPENAME	\$DEVICES	The type name.
\$PAGE	\$DEVICES \$SIGNALS	The schematic page number the object appears on. For signals that connect across pages, this may be any one of the pages, randomly chosen.
\$DEPTH	\$DEVICES	An integer indicating how many layers of hierarchy exist below this symbol. Sorting by this value will produce a "define before use" hierarchical netlist.
\$NUMPINS	\$SIGNALS	The number of pins attached to this net.
\$RAW	\$DEVICES \$SIGNALS	This causes the list to be marked as sorted without sorting it. This is used to merge all items in the list onto one line without performing any sort.
&field	\$DEVICES \$SIGNALS	An attribute field value.

- See also: \$COMBDEVSON/\$COMBDEVSOFF \$COMBPINSON/\$COMBPINSOFF, and "Sorting and Merging" on page 23.

Keyword	150 \$SPACE
Status	
Synopsis	\$SPACE(<i>numCols</i>)
Returns	N.A.
Type	Definition
Where	Top level
Description	Specifies a column spacing for repeated items used when \$ALIGNCOLSON has been enabled. Default is 16. This command will work well if repeating are on a separated line.
Examples	\$ALIGNCOLSON \$SPACE(32)

- See also: \$ALIGNCOLSON/\$ALIGNCOLSOFF

Keyword	151 \$SYSPIN
Status	
Synopsis	\$SYSPIN
Returns	Decimal integer
Type	Data
Where	Script – pin
Description	Returns the systems internal pin sequence number for the current pin. This is valid only for pins actually defined on the device symbol on the schematic and is number from 1 to N. For pins added by a \$SIGSOURCE, this number will be zero.
Examples	<pre>\$DEVPINFORMAT \$PINNAME-\$SYSPIN \$SORT \$DEVICES \$TYPENAME \$COMBDEVSON \$DEVICES\$TYPENAME \$PINS</pre> <p>This script will produce a listing of the pins defined on each device type in the design.</p>

➤ See also: \$PINNUM \$PINSEQ

Keyword	152 \$SYSTEMOPEN
Status	
Synopsis	\$SYSTEMOPEN(filePathandName,["argument1;argument2;... "])
Returns	N.A.
Type	Definition
Where	Top level
Description	Asks the Finder to open the specified file.
Examples	\$SYSTEMOPEN(empt/emptopt.exe)

The \$SYSTEMOPEN command provides a mechanism of starting up other applications from within a script and requesting that document files, such as report output, be opened. The specified file can be either an application or a document. In either case, the action is as if the user double-clicked on the file in the Windows Explorer. If it is an application, it will be launched, or, if it is already running, brought to the front. If it is a document, the application specified as its "creator" will be asked to open it.

➤ See also: \$CREATEREPORT

Keyword	153 \$TAB
Status	
Synopsis	\$TAB
Returns	A tab character.
Type	Data
Where	Script
Description	This keyword is intended to make the intent of a script more clear by using an explicit keyword to insert a tab instead of a literal tab character, which may be misinterpreted by a human reader of a script as one or more blanks.
Examples	Device Name\$TAB\Value \$DEVICES\$DEVNAME\$TAB&Value This creates a simple device listing with 2 columns separated by a tab character.

Keyword	154 \$TABFIELDSON/\$TABFIELDSONOFF
Status	
Synopsis	\$TABFIELDSON \$TABFIELDSONOFF
Returns	N. A.
Type	Definition
Where	Top level
Description	This command specifies that repeating data items should be separated by a tab character. The default is OFF.
Examples	\$TABFIELDSON

➤ See also: \$TAB \$ITEMSEPARATOR \$ALIGNCOLSON/\$ALIGNCOLSONOFF

Keyword	155 \$TABLE
Status	
Synopsis	\$TABLE(blockName[, prefixString[, suffixString]])
Returns	Text
Type	Data
Where	Script
Description	Evaluates a list of IF THEN ELSE conditions specified in the block. If prefixString is specified, it is inserted at the front of the generated string only if the table generates some non-null text. Similarly, if suffixString is specified it will be appended to the generated string if any non-null value is produced.
Examples	<pre> \$DEFINEBLOCK(SYMPParams) &X_BLKNM ,BLKNM=&X_BLKNM &X_HBLKNM ,HBLKNM=&X_HBLKNM &X_LOC ,LOC=&X_LOC &X_LOC_NOT ,LOC<>&X_LOC_NOT \$END \$DEVICES\$TABLE(SYMPParams) </pre> <p>Every time the \$TABLE command is encountered (i.e. for each device in this case), all the lines in the block are evaluated as follows: The item in the first column (i.e. up to the tab character) is evaluated. If it produces any non-null string, the item in the second column is evaluated and output. If the first column produces a null string, the third column (if any) is evaluated and output. Note that the columns are separated by a single tab character.</p>

The \$TABLE command is intended as a shortcut format for specifying a long sequence of IF/THEN/ELSE conditions. It was created for use in netlist formats that may have a large number of optional fields that depend on attribute values specified in the design. Instead of a very long line of \$IF/\$ELSE/\$END sequences, the \$TABLE allows optional fields to be laid out in a more readable vertical format.

The format of a table is as follows:

```

$DEFINEBLOCK(blockName)
    testExpression1tabtrueExpression1tabfalseExpression1
    testExpression2tabtrueExpression2tabfalseExpression2
    testExpression3tabtrueExpression3tabfalseExpression3
    .
    .
    .
$END

```

The tab notation indicates a tab character. Each *testExpression* can be any combination of commands that produces a string. As shown in the above example, this will typically be "&attrField" to test for the existence of an attribute field, but any other expression can be used. Although it is legal, it normally won't make sense for *testExpression* to contain any literal characters, since this would result in the expression always being non-null, or "true".

trueExpression similarly can be any combination of commands and literal characters. If *testExpression* is non-null this expression is evaluated and the resulting string is concatenated to the output. The second tab character and falseExpression are optional. If present, falseExpression is evaluated and the result concatenated if testExpression is null (false).

Note that all lines in the table are evaluated and the results concatenated in the order they are found. The \$TABLE (at least, if used without the optional prefix and suffix arguments) is exactly equivalent to:

```
$IF(testExpression1>trueExpression1$ELSEfalseExpres-  
sion$END$IF(textExpression2) ...
```

The tabs between the columns of the table and the newline characters at the ends of the lines are never included in the output. A tab can be included in any of the expressions by escaping it with a backslash, and a newline can be included by using the \$NEWLINE command.

The \$TABLE command allows for the optional inclusion of a prefix and suffix string. This can allow for a separator character, such as a comma or tab, to be inserted before or after any text generated by the \$TABLE. Note that a comma must be escaped with a backslash in this usage so that it is not confused with an argument terminator. For example:

```
$TABLE(ArgList, \,)
```

If the backslash was not used, the comma would be taken as an argument separator and this would be interpreted as specifying null strings for the prefix and suffix.

Keyword	156 \$TEMPPATH
Status	
Synopsis	\$TEMPPATH
Returns	Text
Type	Data
Where	Script – design
Description	Returns the default Windows temporary path.
Examples	My temporary directory is: \$TEMPPATH

Keyword	157 \$TEXTLINE
Status	
Synopsis	\$TEXTLINE
Returns	Text
Type	Data
Where	Script
Description	Returns the current line read from a text file using the \$INCLUDE \$EXECUTE option. Used in conjunction with the \$EXECUTE option of the \$INCLUDE command, this keyword represents the value of the latest line of text read from the file, not including the line terminator.
Example	<pre> \$DEFINEBLOCK(LineBlock) \$NULL(\$REXEXP((\w)\s+(\S+), \$TEXTLINE)) \$FIND \$DEVICES \$EQ(\$DEVNAME, &1) \$DEVICES\$SETATTR(Value, &2) \$END \$INCLUDE(\$DESIGNNAME.BAK) \$EXECUTE(LineBlock) </pre> <p>The block LineBlock will be executed once for each line in the specified include file. During the execution of this block, the \$TEXTLINE keyword represents the value of the line just read from the file. In this case, the regular expression command is used to extract two text fields from each line. The first field is used to locate a device by name and the second field is used as a new setting for the Value attribute field.</p>

➤ See more information in the \$INCLUDE command

Keyword	158 \$TIME
Status	
Synopsis	\$TIME \$TIME(formatString) \$TIME(formatString,valueString)
Returns	Text
Type	Data
Where	Script
Description	The first form returns the current time of day in the default format. If a first argument is provided, it is taken as a format specification and the time and date are displayed in that format. If a second argument is provided, it is taken as the date value integer to use for conversion.
Example	<pre> \$DEVICES\$DEVNAME was created on \$TIME(\$m/\$d/\$y at \$h/\$n, &DateS- tamp.Dev) </pre> <p>This will output the date and time represented by the date stamp on each device.</p>

When used without an argument list, \$TIME generates the current time of day in the default "long" format for the host machine. This behaviour can be modified by adding an argument string containing format keywords for the various time elements that are available. Any characters in the format string that are not recognized as one of the following items will be included literally in the output string. If a \$ character is needed in the output, it can be escaped by preceding it with a backslash.

\$DATE and \$TIME Format Codes

Code	Meaning
\$M	Month, long (text) form
\$m	Month, numeric form
\$D or \$d	Day of the month, 1..31
\$y	Year, two digits
\$Y	Year, four digits
\$h	Hour, 24-hour format
\$H	Hour, 12-hour format
\$N or \$n	Minute, 00..59
\$S or \$s	Second
\$R or \$r	Raw date value
\$P	AM/PM long form
\$p	AM/PM short form
\$W	Day of week, long form
\$w	Day of week, short form

If second argument is provided, it is taken as a decimal integer raw date value. These values represent the number of seconds since January 1, 1904 and are used to store dates throughout EMTPWorks. For example, the file modified and created dates, device and circuit date stamps are stored in this format. These can be converted to human-readable format by inserting them as a second argument to \$TIME.

Note that \$DATE and \$TIME are identical in function when used with arguments.

➤ See also: \$DATE \$TIMECREATED/\$TIMEMODIFIED

Keyword	159 \$TIMECREATED/\$TIMEMODIFIED
Status	
Synopsis	\$TIMECREATED \$TIMEMODIFIED \$TIMECREATED(<i>formatString</i>) \$TIMEMODIFIED(<i>formatString</i>)
Returns	Text
Type	Data
Where	Script
Description	If used without arguments, \$TIMECREATED and \$TIMEMODIFIED return the created or modified time of day of the current design in the default format. An argument list can be added to specify any time or date format. When used with an argument list, \$TIMECREATED and \$DATECREATED are identical in function and, similarly, \$TIMEMODIFIED and \$DATEMODIFIED are identical.
Example	Created at \$TIMECREATED on \$DATECREATED

These two command are variations of the \$TIME command and behave identically except that they use the created or modified date of the current design, rather than the current date.

➤ See also: \$DATECREATED/\$DATEMODIFIED

Keyword	160 \$TYPENAME
Status	
Synopsis	\$TYPENAME
Returns	String
Type	Data
Where	Script - device
Description	Returns the type name of the current device, that is, the name of the type definition as it would appear in the parts palette.
Example	\$SORT \$DEVICES \$TYPENAME \$COMBDEVSON \$DEVICES\$TYPENAME \$DEVNAME This will generate a simple bill of materials, sorted and merged by type name.

For most "bill of materials" listings, the Part attribute should be used instead of \$TYPENAME. In the standard EMTPWorks libraries, the Part field contains the actual manufacturers part number, whereas the type name may be contracted or include some gate packaging information. In addition, the type name cannot normally be edited in a schematic, so it gives less flexibility.

Keyword	161 \$UNCONNPINSOFF/\$UNCONNPINSON
Status	
Synopsis	\$UNCONNPINSOFF \$UNCONNPINSON
Returns	N. A.
Type	Definition
Where	Top level
Description	When ON, allows device pins that have no signal lines attached to them and are not connected to any other signals by name to appear in the netlist, otherwise they are suppressed. Default is ON.
Example	\$UNCONNPINSOFF

➤ See also: \$ISUNCONNPIN

Keyword	162 \$UNNAMEDDEVS
Status	
Synopsis	\$UNNAMEDDEVS(<i>string</i>)
Returns	N. A.
Type	Definition
Where	Top level
Description	This command sets the string to be output when \$DEVNAME is referred to for an unnamed device. The default is "unnamed".
Example	\$UNNAMEDDEVS() This specifies that nothing (a null string) should be output when the name of an unnamed signal is requested.

Keyword	163 \$UNNAMEDSIGS
Status	
Synopsis	\$UNNAMEDSIGS(<i>string</i>)
Returns	N. A.
Type	Definition
Where	Top level
Description	This command sets the string to be output when \$SIGNAME is referred to for an unnamed signal. The default is "unnamed".
Example	\$UNNAMEDSIGS() This specifies that nothing (a null string) should be output when the name of an unnamed signal is requested.

Keyword	164 \$UNSELECTEDPINS
Status	
Synopsis	\$UNSELECTEDPINS [\$ON \$OFF] [\$WARN \$NOWARN]
Returns	N. A.
Type	Definition
Where	Top level
Description	This command determines what action is taken when an attempt is made to list a pin which is linked to a signal that is not selected for listing. The default is \$ON \$WARN.
Example	\$UNSELECTEDPINS \$OFF This specifies that any pins that are attached to signals that are not selected for listing should be skipped.

When the Scriptor generates a netlist report from a design, it makes its own internal list of the devices and signals to be included. The list takes into account the hierarchy mode of the report and any \$FIND commands that have been executed. It is possible using \$FIND commands to create a situation in which you are creating a \$DEVICES \$PINS listing (e.g. a SPICE-format netlist) which is requesting information about a pin that is not in any selected signal.

The \$ON/\$OFF setting lets you choose whether pins on unselected signals are included in the listing. Setting this to \$OFF means that such signals are not included. The default is \$ON.

The \$WARN/\$NOWARN setting determines if a warning is issued if an attempt is made to extract data about a signal that is not selected. The default is \$WARN.

The setting \$UNSELECTEDPINS \$OFF \$NOWARN is primarily useful in cases where it is desired to completely omit some device pins if they are not attached to anything in the schematic. This can be useful especially in FPGA netlists where logic can be omitted if not used.

Keyword	165 \$UNUSEDUNITS
Status	
Synopsis	\$UNUSEDUNITS
Returns	Text
Type	Data
Where	Script - Design
Description	Generates a text list of the free gate units in the current design. There is no control over the format of this listing.
Example	\$UNUSEDUNITS This will generate a listing like this one: U12 74ALS00DN a c U34 74ALS1120 b c etc.

This command produces a text string which enumerates the gate units in the Packager's free gate table. This type of report can be used to optimize use of gate packages or to provide documentation on the schematic itself of unused units.

See more information on packaging in the chapter "Device Packaging and Naming" in the EMTPWorks User's Guide.

Keyword	166 \$UPPERCASE
Status	
Synopsis	\$UPPERCASE(<i>string</i>)
Returns	String
Type	Data
Where	Script
Description	Converts all alphabetic characters in the argument to upper case. Non-alphabetic characters are not changed..
Example	\$UPPERCASE(\$DEVNAME)

➤ See also: \$LOWERCASE

Keyword	167 \$VERIFY
Status	
Synopsis	\$VERIFY(blockName, string)
Returns	Boolean
Type	Data
Where	Script
Description	Used to determine if a given string value is in a list of allowable values. Returns TRUE if the given string is in the mapping table.
Example	<pre> \$DEFINEBLOCK(GoodPkgs) DIP14 DIP16 DIP20 \$END \$IF(\$NOT(\$VERIFY(GoodPkgs, &Package)))Bad package!\$END </pre>

This command allows you to determine if a given string value exists in a table of values. The table format is exactly the same as that used for the \$MAP command but in this case the second column of the table (if specified) is ignored. You can therefore use the same table in two different places, once for error checking and elsewhere for actually mapping an output value.

- See the \$MAP keyword for the table format.

Keyword	168 \$WRITETRANSRIPT
Status	
Synopsis	\$WRITETRANSRIPT(string)
Returns	N.A.
Type	Definition
Where	Script
Description	Writes the specified string to the current transcript file.
Example	\$WRITETRANSRIPT(We got &ErrCnt errors!)

This command writes the specified string to the current transcript file. Note that the string argument can be a literal string or any combination of script commands that generates a string.

If there is no current transcript file (i.e. if no \$CREATE TRANSCRIPT has been done), no output is generated and this command is ignored.

- See also \$CREATETRANSRIPT and “File Input and Output” on page 37.