DElight for intermediates

by

B. Nye and D. Wang

Memorandum No. UCB/ERL M85/32

26 April 1985
DELIGHT FOR INTERMEDIATES

by

B. Nye and D. Wang

Memorandum No. UCB/ERL M85/32

26 April 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720
DELIGHT FOR INTERMEDIATES

by

B. Nye and D. Wang

Memorandum No. UCB/ERL M85/32
26 April 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720
The purpose of this guide is to provide additional information for users already familiar with the basic DELIGHT features covered in *DELIGHT For Beginners*. DELIGHT, an interactive optimization-based computer-aided design system designed to provide a friendly and flexible environment for designers working in a multitude of disciplines, has evolved greatly since the publication of the Beginners Guide. The most notable of sections included here are one discussing the online help system, a thorough survey of debugging techniques, and one documenting how new application-specific DELIGHT versions are created and tested. Throughout this guide references to additional information available using this online help system have also been included. Through judicious use of this guide, DELIGHT users should be able to take advantage of this accrual of pragmatic and productive computer-aided design diversity.
# Table of Contents

1 Introduction ......................................................................................... 4

2 Online Help System ........................................................................... 5
   2.1 Introduction ................................................................................. 5
   2.2 Using the Help Commands ............................................................ 5
   2.3 Where Binary Help Files are Found ............................................. 8

3 More Define Enhancements ................................................................. 9
   3.1 Review of Define Enhancements .................................................... 9
   3.2 No-Quote Convention ................................................................. 10
   3.3 Define Options .......................................................................... 11
   3.4 Auto-Pushback Convention ......................................................... 15
   3.5 Creating New Commands With Defines ...................................... 17

4 The New Plot Options ...................................................................... 18

5 Additional I/O Features .................................................................... 22
   5.1 Writing Interactive Programs With Answer_to_prompt ............... 24
   5.2 File Input and Output ................................................................. 27
      5.2.1 The Openhdtl File .............................................................. 27
      5.2.2 File I/O With Built-in Functions ........................................... 29
      5.2.3 Opening Temporary Scratch Files With Opuniq .................. 31
      5.2.4 An Application of Opuniq ................................................. 33

6 Language Extensibility Using Macros .............................................. 35
   6.1 Tokens and Push-Back ............................................................... 35
   6.2 Language Extensibility Using Macros ........................................ 38

7 Debugging Rattle Programs ............................................................. 40
   7.1 Debugging Compiler-Reported Errors ........................................ 41
      7.1.1 Tracing What is Pushed Back ............................................... 42
      7.1.2 Other Debugging Suggestions .............................................. 43
   7.2 Debugging Run-Time Errors ...................................................... 44
      7.2.1 What Run-Time Errors Are .................................................. 44
      7.2.2 Review of Commands for Debugging Run-Time Errors ........ 45
      7.2.3 Using Pdebug_ For Debugging ............................................ 46
      7.2.4 Debugging by Adding Print and Suspend Statements ............ 48
      7.2.5 Aborting On Numeric Overflow .......................................... 50
      7.2.6 DELIGHT Internal Aborts and the Hardreset Command .......... 51
   7.3 General Use of Whatis and Whereis ............................................ 52

8 Creating New DELIGHT Versions ..................................................... 54
   8.1 Adding Built-in Routines .......................................................... 55
   8.2 Declaring Variables for Rattle Access .......................................... 58
   8.3 Version-Specific Routines Called by DELIGHT ......................... 60
   8.4 Loading DELIGHT .................................................................. 64
   8.5 Making a Memfile ..................................................................... 64
   8.6 Starting DELIGHT ................................................................... 69
   8.7 Debugging Added Built-in Routines .......................................... 73
   8.8 General Guidelines for Creating a DELIGHT Version ............... 74
1 Introduction

The DELIGHT For Intermediates guide has two purposes. One is to point out more advanced features for users who are already familiar with the basic ideas presented in the DELIGHT For Beginners guide [4]. The second is to present several important new features that have come into being since the publication of the Beginners Guide. For a complete discussion of many more technical details than presented here, see the Ph.D. Dissertation of W.T. Nye [5].

This guide makes liberal use of examples, presented in the form of terminal dialogue. In this dialogue user input is in boldface for clarity. Also, blank lines have been occasionally inserted at various places in the terminal dialogue to separate groups of statements, and will not appear on the terminal screen\(^1\).

As a reminder, a backslash ("\") at the end of a line indicates that the line is to be continued onto the next—that the fictitious newline character at the end of the line has been "escaped" (had its meaning changed) so that it no longer terminates the line containing it. The DELIGHT prompt for the continued line changes to "1\", "2\" etc. as seen in the following examples:

```
1> print 1.2
1 \ 34
1 .234
1> suspend
2> print 1.2
2 \ 34
1 .234
2> reset
1>
```

The plan of this guide is as follows. We begin with the usage, descriptions and examples of the online help facility commands in section 2. Section 3 discusses enhancements to Rattle defines. After a review of defines, it then proceeds to the various enhancements including, probably the most important, define options. Finally, the last subsection illustrates the general idea of creating new commands using defines, define options, and Rattle procedures. Section 4 shows the power of define options by demonstrating all the nifty new options associated with the plot command. Additional I/O features are covered in section 5. One, for example, is an easy way to write Rattle procedures that interact with the user through question/answer dialogue. Another topic addressed is how file input/output (I/O) is accomplished in DELIGHT. Just as the Beginners Guide showed how defines are used for Rattle extensibility, section 6 presents extensibility using Rattle macros by first explaining the concepts of tokens and the DELIGHT push-back mechanism. Debugging facilities for compile-time and run-time bugs are introduced in section 7. Finally, section 8 considers the entire process of creating new application-specific DELIGHT versions. It explains how to add built-in routines, declare variables for Rattle access, load/link the executable program, make a new memfile, and start the new DELIGHT version.

---

\(^1\) When using DELIGHT on the UNIX (a trademark of Bell Laboratories) operating system (and possibly on other systems), there is a command called helper which may be used to obtain the same on-line assistance available in DELIGHT—and with exactly the same command syntax. These commands will be explained in section 2 of this guide; additional information is available in [6].
2 Online Help System

2.1 Introduction

This section introduces users to the help facility available in DELIGHT. Through various easy-to-use commands, quick on-line assistance is made available for DELIGHT commands, features, topics, tutorials, etc.—basically, for whatever information has been set up by system personnel or even by other users. Moreover, this on-line assistance is obtained quickly—even if there are many help entries available—since the large binary help files read by the help commands are "hashed" for relatively rapid table lookup.

The help commands fall into four categories: (1) the basic help command for displaying all the standard parts (fields) of a help entry and the helpall command for displaying all fields, (2) the helpsubject command for showing a brief description of all help entries having to do with a specified subject, (3) the helpnewer command for displaying commands newer than a specified date, i.e., according to when they were created, and (4) several commands for displaying certain fields of a help entry quickly. These commands are briefly summarized in the following table:

<table>
<thead>
<tr>
<th>Online Help Commands</th>
<th>What is Displayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>All standard help entry fields.</td>
</tr>
<tr>
<td>helpall</td>
<td>All help entry fields.</td>
</tr>
<tr>
<td>helpsubject</td>
<td>A brief description of all entries having to do with a given subject.</td>
</tr>
<tr>
<td>helpnewer</td>
<td>All commands or options newer than a given month-year.</td>
</tr>
<tr>
<td>helpexamples</td>
<td>Just the EXAMPLES field.</td>
</tr>
<tr>
<td>helpnext</td>
<td>Just the NEXT field.</td>
</tr>
<tr>
<td>helpoptions</td>
<td>Just the OPTIONS field.</td>
</tr>
<tr>
<td>helpusage</td>
<td>Just the USAGE field.</td>
</tr>
</tbody>
</table>

The remainder of this section consists of two subsections: 2.2 gives the usage, description and examples of help commands while 2.3 shows where the binary help files reside that the help commands open and read.

2.2 Using the Help Commands

To get assistance on commands, features, topics, tutorials, etc. while inside DELIGHT, type help, followed by the particular command or topic name of interest. For example you can get information on the DELIGHT output_to command by typing help output_to:
6 DEIGHT For Intermediates (3/1/85)

1> help output_to

NAME
output_to - Make all following DELIGHT output go to a file.

USAGE
output_to FILENAME

OPTIONS
~verbose=YES YES prints message about creation/overwrite/append of file.

EXAMPLES
output_to diary
output_to ~ verbose diary

SEE ALSO
outputonto, output_end, echo_to, ?

If you do not know what help entries (by name) are available, the help command is set up so that typing help alone has the same effect as typing help help and shows how to use online help. Don’t worry about the OPTIONS field above if you don’t understand what they are; they will be explained in detail later in section 3. Suffice it to say that in the above example, the ~verbose option is shown followed by =YES, indicating that the option has a default setting of YES and that thus, the messages are printed. How to turn off this option for a particular use of the command is shown in the second example above.

As just mentioned, one of the problems with an online help facility which provides help by command or topic name is that users do not know what commands or topics are available. For this reason, DEUGHT provides the helpsubject command for showing a brief description of all help entries having to do with a given subject. For example, try the following:

1> helpssubject draw
clip_vector - Draw vector between 2 coordinates, clipping to viewport.
clip_draw - Draw vector to x,y coordinate, clipping to stay in viewport.
clip_move - Position beginning of a vector, ready for a clip_draw.
draw - Draw vector from previous position to specified x,y coord.

In the above, you have been presented with four commands that are related to the subject of drawing graphics vectors. After seeing this output, one would probably pursue additional information on one of the commands as in:

1> help draw

NAME
draw - Draw vector from previous position to specified x,y coord.

USAGE
draw X Y

EXAMPLES
draw .2 .5
draw xorig=x yorig=y

SEE ALSO
move, clip_draw, <graphics>

The listing <graphics> under SEE ALSO above indicates a subject area instead of a command, procedure, define, etc., and its help entry may still be obtained in the usual way, i.e., by typing help <graphics>.
The `helpsubject` command is set up so that from 1 to 6 subjects can be explored with the same `helpsubject` command. The help entries listed are those having to do with either the first subject or the second or the third, etc., as seen in the following:

```
1> helpsubject move
clip_move - Position beginning of a vector, ready for a clip_draw.
move - Position beginning of a vector, ready for a draw.
```

```
1> helpsubject move draw
clip_move - Position beginning of a vector, ready for a clip_draw.
move - Position beginning of a vector, ready for a draw.
clip_vector - Draw vector between 2 coordinates, clipping to viewport.
clip_draw - Draw vector to x,y coordinate, clipping to stay in viewport.
draw - Draw vector from previous position to specified x,y coord.
1>
```

To get more on how to use `helpsubject`, type `help helpsubject`. To see a list of all available online help entries, type `helpsubject *`.

Frequently an occasional DELIGHT user wishes to know what new commands or features have been recently added to the system. The `helpnewer` command allows you to see a list of all new help entries, i.e., of all entries that have been added to the online help system since a specified date. If DELIGHT system personnel have been consistent in creating or updating help entries, these entries should represent everything new that has been added to DELIGHT since the given date. The date is specified as a numeric month-year pair. For example, you can type `helpnewer 8-84` to see all help entries that were added from August, 1984 to the present (assumed to be October, 1984 in the following example). Note that the following output to this command has been shortened. In fact, as new commands or features are added to DELIGHT and their corresponding help entries added to the online help system, the output actually seen when working through this guide may be considerably larger!

```
1> helpnewer 8-84
SYSTEM, BASIC HELP (file "$HsBASL0"):
8-84:
printvs - Print the values of from 1 to 6 expressions in column format.
plot - (NEW OPTIONS)
  ^verbose=NO If YES, causes the message "--- Compiling plot loop ---".
  ^xorigin=0.0 X world coordinate value that "xorigin" causes the axes
  ^yorigin=0.0 Y world coordinate value that "yorigin" causes the axes
output_end - (NEW OPTIONS)
  ^verbose=NO YES means print "Output is in FILENAME".
8-84:
LINPROG - Solve a linear program.
difast - Turn on or off "Fast Rattle" execution.
10-84:
run - (NEW OPTIONS)
  ^suspend=NO If set to YES, execution interrupts after NUMBER
printv - (NEW OPTIONS)
  ^MaxSig=6 The maximum number of significant figures printed.
  ^MinSig=0 The minimum number of significant figures printed.
1>
```

The above output shows that `helpnewer` presents the new commands and features by month and indicates that in August, 1984, the `printvs` command was created, the `plot` command got three new options, and the `output_end` command got one new option. Similarly, in September, 1984, the `LINPROG` command and `difast` feature were created.
A common occurrence in working with a program containing many commands is the need to review quickly the syntax of or see examples of how to use a command. Similarly, the options and their default settings may need to be reviewed even though a user is familiar with how to use a command. Of course, this information can be obtained using the \texttt{help} command but \texttt{help} usually produces too much output. For this reason, there are several commands that just print out certain fields of a help entry. The \texttt{helpexamples}, \texttt{helpoptions}, and \texttt{helpusage} commands display, respectively, the \texttt{EXAMPLES}, \texttt{OPTIONS}, and \texttt{USAGE} fields of a help entry, as seen in the following:

\begin{verbatim}
1> helpexamples vector
   EXAMPLES
   vector .2 .5 1 1
   vector xorig yorig Wx (2*Wy - 0.5)
1> helpusage plot
   USAGE
   plot YEXPR1 [ YEXPR2 ... ] vs XVAR from EXPR to EXPR [ by ] EXPR
   [ times ] EXPR
   [ out ]
   [ dec ]
   [ log ]

1> helpoptions output_to
   OPTIONS
   ~verbose=YES YES prints message about creation/overwrite/append of file.
1>
\end{verbatim}

Another command that prints out just one field is \texttt{helpnext}, which prints the \texttt{NEXT} field of a help entry. This field could contain suggestions for the next thing to do after issuing a command. This might be useful, for example, in a design procedure that contained many steps.

\section{2.3 Where Binary Help Files are Found}

When one of the help commands is executed, it must open and read from a binary help file, containing help entries, that has already been set up. The system tries to open four different files, if they exist. Assuming the DELIGHT version is \texttt{XXXX} (\texttt{BASIC} for the basic DELIGHT system, \texttt{MIMO} for DELIGHT.MIMO, \texttt{SPICE} for DELIGHT.SPICE etc.—see section 7) the binary help files are tried in the order shown:

1. \texttt{HXXXXX} - Private, (local) version-specific help file
2. <\texttt{HXXXXX}> - Shared, (local) version-specific help file
3. <\texttt{HsXXXXX}> - System, version-specific help file
4. <\texttt{HsBASIC}> - System, basic help file

The first file tried exists in the user's current directory and is set up according to the rules in [6]. It allows the user to have available help assistance that he created himself for version \texttt{XXXX} of DELIGHT. The second file tried is for help assistance that is to be shared by several users working with version \texttt{XXXX} of DELIGHT. The third file tried is for help entries that are to be shared by all users working with version \texttt{XXXX} of DELIGHT. Finally, the fourth file tried contains help assistance that is basic to any DELIGHT version. Since this file is usually set up by system personnel, it is named <\texttt{HsBASIC}>. For the time being, simply treat the brackets "<" and ">" as part of the filename. They indicate that the files are located in another directory and section 5.2.1 of this guide is explicitly directed towards filenames of this sort and where the files are found. Notice the first line after typing a \texttt{helpnewer} command:
This line indicates that the following entries are from file `<HsBASIC>`.
The system tried to open the other three files in the table above but they did not exist. If they had, the line

```
SYSTEM, BASIC HELP (file "<HsBASIC>"):
```

for example, might have been seen with all of its "newer" entries by month, followed by the output above.

Advanced users who wish to know more about help commands or wish to set up their own binary help files (such as file `<HsSPICE>` in the above table) should see the document `The Helper Facility` [6], which describes a general purpose online help facility called `helper` that is not particular to DEUGHT. However, all of the help commands in `helper` also exist in DEUGHT.

### 3 More Define Enhancements

This section begins with a review of the basic define features discussed in `DELIGHT For Beginners`. This is followed by the more advanced `no-quotes` and `auto-pushback` conventions in sections 3.2 and 3.4, respectively. Section 3.3 introduces a very important new feature of defines, define options. Section 3.5 then gives suggestions regarding the creation of new commands with defines and Rattle procedures, one of the cornerstone features of DEUGHT.

#### 3.1 Review of Define Enhancements

This section reviews the basic define features presented in the Beginners Guide. Recall that the simplest usage of defines is to substitute one piece of text for another. For example,

```
define (TWOP,6.283185307)
```

allows you to easily use the value of $2\pi$ in expressions. Next we extended defines to have arguments such as $x$ in

```
define (print_square x,print x**2).
```

Then, to allow a define to be more readable, special keywords (literal strings) were allowed in between arguments such as `over` in the define

```
define (print_ratio x 'over' y,print x/y).
```

which, for example, could be invoked using `print_ratio 5 over 2`.

The next important extension to defines was to allow optional arguments having
default values. These arguments come after a semicolon (";") and have their default values following an equal sign such as \texttt{xscale} in the define

\begin{verbatim}
define (print_scaled x ; xscale=1 , print x/xscale).
\end{verbatim}

Another extension was multiline defines, which do not have leading left parentheses and end with the keyword \texttt{end}. For example,

\begin{verbatim}
define NewtonUpdate x
  x = x - f(x)/deriv(x)
print x
end
\end{verbatim}

is a define whose definition consists of two lines of Rattle.

Finally, \textit{DELIGHT For Beginners} demonstrated the \textit{double-quote convention} in which a define argument preceded by two consecutive single quotes means to quote the substitution string for that argument before substituting it into the define definition. For example,

\begin{verbatim}
c\define (list '"name, list_(name))
\end{verbatim}

causes \texttt{list myfile} to be substituted by \texttt{list ("myfile")}; the \texttt{name} argument value \texttt{myfile} has been quoted before being substituted where \texttt{name} occurs in the definition.

### 3.2 No-Quote Convention

There are several cases in which you may want to switch off the double-quote convention in using a define which was originally created with an argument preceded by two quotes. Suppose you want to create a (rather silly) procedure to list a file then edit the file using the \textit{DELIGHT} built-in editor. You could use the following:

\begin{verbatim}
1> define (ListEdit '"name, Lproc(name))
1> procedure Lproc (pnone)
1|    list pnone
1|    edit pnone
1>
\end{verbatim}

However, upon trying your \texttt{ListEdit} command (first, creating a dummy file as shown):

\begin{verbatim}
1> edit junk
Unable to open "junk"
:a
Inside file junk
.
.
.:wq
"junk" 1 lines
1> ListEdit junk
ERROR: list: Cannot open "pnone"
Unable to open "pnone"
:a
1>
\end{verbatim}
you discover that the system thinks the file you want to list and edit is file \textit{pname} instead of file \textit{junk}. The problem is that the \textit{list} and \textit{edit} commands are taking your argument \textit{pname} literally; \textit{list} and \textit{edit}, like most other "commands", are actually defines which, in this case have arguments that are preceded by two quotes, something like \texttt{define (list 'name, list (name))}. What we want the \textit{list} and \textit{edit} in the above procedure \texttt{LEproc} to do is to take their arguments from variable \textit{pname} instead of taking \textit{pname} literally. For this \textsc{DEUGHT} once again extends the list of define features with the no-quote convention.

The convention is that a define argument originally to be quoted using the double-quote convention may turn off the double-quote convention for a single use of the define—the argument’s value may be substituted into the definition without surrounding quotes—by preceding the actual argument with the character "<", meaning, take the argument literal from the contents of the argument variable. Thus you could redefine procedure \texttt{LEproc} and test \texttt{ListEdit} as follows:

\begin{verbatim}
\$ procedure LEproc (pname) {
| list <pname
| edit <pname
\}
\$ ListEdit junk
----------- Begin junk ---------------
inside file junk
----------- End junk ---------------
"junk" 1 lines
:q
\$ 
\end{verbatim}

This demonstrates that the no-quote convention applied to the two \textit{pname} arguments was successful.

3.3 Define Options

There are many types of commands for which it is quite natural to have choices that are made optionally, i.e., that have things which you may or may not wish to set or choose. For example, one can imagine many possibilities with a \texttt{plot} command—whether or not to erase the screen, whether to use logarithmic axes, what values to force the x and y axes limits to be, etc. Another example is having the possibilities of turning on line numbers with the \texttt{list} command and turning on or off the rows of dashes that begin and end the file listing. Using optional arguments (those following the semicolon in a define declaration), you could define \texttt{list} as

\texttt{define (list 'name ; linenumbers=NO dashes=YES , ...}

with one required argument (the filename) and two optional arguments (the two YES/NO choices). Then \texttt{list} could be used as any of the following ways:

\begin{verbatim}
list myfile
list myfile YES
list myfile YES NO
list myfile NO NO
\end{verbatim}

Obviously, there are several objections that can be raised. One is that you will probably forget the order of the two optional arguments and have to keep referring to online help. Another is that this statement is not self-documenting—if one of the last three
lines above appears in a procedure, someone looking at the source code cannot easily
tell what the statement is supposed to do since the meaning of the YES/NO arguments
may be forgotten. Finally, to set the dashes argument, the linenumbers argument
must be set first, as in the last example above. In other words when using the define,
arguments must be specified in the same fixed order that they were specified when the
define was declared.

To allow greater flexibility in using a command with optional things that may be
specified, defines are hereby extended to allow options—not to be confused with
optional arguments. Option names are preceded by a tilde ("\~") and must come
directly after the define name (before any arguments) in the define declaration.
Option names, just as for arguments, may appear anywhere in the definition and just
represent places where text gets substituted. Also, just as for optional arguments, the
default value of an option is placed after an equals sign ("\=") following the option name.
Recall the define declaration define (print_scaled z ; zscale=1 ,print z/zscale) from
section 3.1. Try the following simple (though not very useful) redefinition and use of
print_scaled:

```plaintext
define (print-scaled ~xscale=1 x, print x/xscale)
define (print_scaled ~xscale=2 5
  2.500
define (print_scaled ~xscale=-10 5
  -1.000e+3
```

Notice that in the declaration the one option ~xscale comes directly after the define
name print_scaled, before the one required argument x. Also notice that zscale
appears in the definition (after the comma) in any manner just as z and that when the
command is used, zscale in the definition gets replaced by its default value of 1.

By the very nature of options, the ability to set their values is essential. This is done
when using a define by following the define name by tilde, the option name, an equals
sign, and the option's value. This can be seen in the following continuation of the
above:

```plaintext
define (print_scaled ~xscale=2 5
  2.500
define (print_scaled ~xscale=-10 5
  -1.000e+3
```

By looking at the definition for print_scaled above, it is clear that the second example
above gets substituted by print 5/(-10).

Let us try a more sophisticated example—one with more than one option:
This multiline define is a print statement with options to place stars in front of or to underline the printed output. By default, the stars are printed but the underline is not. Recall from *DEIGHT For Beginners* that the go statement is needed to prevent DEIGHT from awaiting a possible else clause to the last if-statement above. Below are examples of the *printfancy* command:

```bash
1> printfancy 5
      **** value = 5.000
1> printfancy ~line=YES 5
      **** value = 5.000
1> printfancy ~stars=NO 5
      value = 5.000
1> printfancy ~stars=NO ~line=YES 5
      value = 5.000
1> printfancy ~line=YES ~stars=NO 5
      value = 5.000
1>
```

The last two examples show that the options can be specified in any order.

In the *print_scaled* define, the option *zscale* could take on any numeric value. But in the *printfancy* define, the two options could take on YES or NO values. It turns out that there are many cases where options take on YES or NO values. To simplify setting options to YES or NO when using a define the following conventions have been adopted: (1) if the option name is NOT followed by an equal sign and value, its value becomes YES; (2) if the tilde is followed by an exclamation mark ("!"） before the option name (and also no equal sign), then the option value becomes NO. Thus, the last use of the *printfancy* command above may be more easily written:

```bash
1> printfancy ~line=! stars 5
      value = 5.000
1>
```

This example shows how to switch on underlining just for a single use of a define. You may, however, wish to always have underlines. This is equivalent to having the default value for option ~line changed from NO to YES. With the set_option command, DEIGHT allows you to change at any time the default value of any option of any define. It has syntax:

```
set_option DEFINE_NAME ~OPTION_NAME=NEW_VALUE
```

and can be used to change the default value for option ~line as follows:
Even though the default value for option `~line` is now `YES` you can still use `printfancy` without underlines:

```
1> printfancy ~line 5
***** value = 5.000
```

To show what happens if you misspell the option name, try the following:

```
1> printfancy ~lines 5
ERROR: For define "printfancy", option "lines" does not exist.
ERROR: Illegal statement: "5"
```

The ability to change the default values for options immediately brings with it the need to be able to display the current default values of define options. For this, the `display` command, first introduced in section 10 of *DELIGHT For Beginners*, allows the argument `doptions` (for define `options`) followed by a define name:

```
1> display doptions printfancy
1 define with options:
    printfancy      ~stars          YES
        ~line          YES
```

```
1> set-option printfancy ~line=NO
1> display doptions printfancy
1 define with options:
    printfancy      ~stars          YES
        ~line          NO
```

Shown are the define name, the option names and their default values.

Let's consider one final example—the practical requirement of getting slightly more information from the `help` command (see section 2.2). We can see what options are available as follows:
The following options are YES/NO flags, along with their default values, of whether to print the indicated help entry field:

- NAME=YES
- USAGE=YES
- DESCRIPTION=YES
- MORE_DETAIL=NO
- OPTIONS=YES
- EXAMPLES=YES
- SEE_ALSO=YES
- BUGS=YES
- SOURCED_FILE=NO
- AUTHOR=NO
- KEYWORDS=NO
- NEXT=YES

The default values shown for the SOURCE_FILE and AUTHOR options are both NO as seen in the following:

```
1> help enter
NAME
   enter - Enter a procedure for examining local variables, etc.
USAGE
   enter [PROCNAME]
EXAMPLES
   enter algo
SEE ALSO
   leave
```

If you wanted to see these fields for the enter command you could type:

```
1> help SOURCE_FILE AUTHOR enter
NAME
   enter - Enter a procedure for examining local variables, etc.
USAGE
   enter [PROCNAME]
EXAMPLES
   enter algo
SEE ALSO
   leave
SOURCE FILE
   <enter>
AUTHOR
   Bill Nye
```

However, if you wanted to always see these fields for all uses of the help command, the following would suffice:

```
set_option help SOURCE_FILE=YES
set_option help AUTHOR=YES
```

### 3.4 Auto-Pushback Convention

The auto-pushback convention—not a terribly important feature—has to do with defines which are never issued as commands but whose only purpose is to have options associated with them so that the values of certain variables can be controlled by setting the default values of the options. This might be useful when you are developing a Rattle program consisting of subprocesses or substeps such as a simulator that have variables or parameters whose values you would like to have a user set by setting the default values of options. The convention simply says that if, in the define declaration,
the define name is preceded by a tilde (~) as in define(~flags...), then any set_option on one of the options of this define will automatically push back the define name. (See section 6.1 if you don't understand what is meant by "push back".) Let's first see what this means with a simple (though not very useful) example:

```plaintext
1> define (~simple ~junk=1 .print -junk)
1> simple
-1.000
1> set_option simple ~junk=2
-2.000
1> simple
-2.000
```

Notice that after the set_option command, the define name simple has been automatically pushed back with the new default value for option ~junk, causing print -2 to be executed. As a check, when simple is typed, the same result is obtained.

To control important variables using options, as mentioned at the beginning of this section, you simply have a define definition contain assignments that use the options. Suppose you want to control two simulator flags using options. The following is a possibility:

```plaintext
1> define ~simulator_flags ~flag1=0 ~flag2=0
1> variable_flag1 = flag1
1> variable_flag2 = flag2
1> end
1> set_option simulator_flags ~flag1=9
1> display variables var*
2 variables:
  variable_flag1  = 9.00000
  variable_flag2  = 0.00000
```

After typing the set_option command, the define name simulator_flags was pushed back causing variable variable_flag1 to be assigned the value 9.

One benefit of using options instead of simply allowing users of your program to set variables directly is the ability to use the display doptions command to see the current option values:

```plaintext
1> display doptions simulator_flags
1 define with options:
   simulator_flags   ~flag1  9
                    ~flag2  0
```

This benefit is even more important when a second benefit of using options is also
considered. This is when the variables such as variable_flag1 above are complicated expressions of the option flags or when the option values are passed to another procedure, as seen in the following hypothetical example:

```plaintext
define simulator_options ~MaxIter=100 ~Algo=trapezoidal
    simulator_options_(MaxIter, quote Algo)
end

procedure simulator_options_(MaxIter, Algo_string) {
    import Tfanax, Tduration, Vmax
    Tduration = Tfanax / MaxIter
    Vmax = exp(-Tduration/70.5)
    set_algo (Algo_string)
}
```

In the above, simulator parameters $T_{duration}$ and $V_{max}$ are expressions of the user-settable option $\sim_{MaxIter}$, while user-settable option $\sim_{Algo}$ is passed to procedure `set_algo`.

### 3.5 Creating New Commands With Defines

One of the cornerstones in how we build up interactive DELIGHT design systems is the idea of using defines and all their extensions to create new commands. While this idea was alluded to in the Beginners Guide, this section will firm up a few practical considerations on the best use of this technique.

The steps a DELIGHT user should follow to create his own commands with defines are:

1. Decide on a command name.
2. Write a procedure to do what the command is supposed to do. A good idea is to have the procedure name be the command name followed by an underscore ("_") as in command `showalgo` and procedure `showalgo_`.
3. Write the define statement so that the definition simply invokes the procedure as in define `(showalgo,showalgo_)`.

A good reason for making the procedure name similar in this way to the command name is so that if an interrupt of some kind occurs while executing inside the procedure, it will be easy to determine what command/statement caused the error from trace output. This idea is demonstrated further below.

In the following, we repeat the `printfancy` command from section 3.3 by using the above define/procedure paradigm instead of the multiline define used before, which was:

```plaintext
define printfancy ~stars=YES ~line=NO X
    if (stars==YES) printf '*****
    printf 'value = $r/n' X
    if (line==YES) printf '---------------------------/n'
    go
end
```

However, to show the trace output that occurs during an interrupt of execution, we
print the reciprocal of the value given as argument.

```lisp
> procedure printfancy(starsflag, lineflag, x) {
    if (starsflag=YES) printf '*****
    printf 'value = %n' 1/x
    if (lineflag=YES) printf '-----------------1/n'
}
> define (printfancy ~stars=YES ~line=NO X,printfancy(stars,line,X))
```

As before, this command could be used in any of the following ways, which yield exactly the same results (except for the reciprocal) as before:

```lisp
> printfancy 5
***** value = .2000
> printfancy ~line 5
-----------------
> printfancy ~! stars 5
value = .2000
> printfancy 5
***** value = .2000
> printfancy 0
*****
RUN-TIME ERROR: 1 overflow(s) or other floating point exception(s).
```

Interrupt...
```
> trace
Interrupted IN procedure printfancy— (Input from the terminal)
> reset
```

The major benefit of creating commands in this way—using procedures instead of multiline defines—is that the procedure body is Rattle compiled just once whereas the definition of a multiline define must be recompiled every time you use the command. As you might have noticed, the `printfancy` command here is much much faster than the one set up in section 3.3.

4 The New Plot Options

Both as an example of how define options have been used and as valuable examples in their own right, this section demonstrates the new options associated with the `plot` command. Since the simple usage of `plot` was already demonstrated in the Beginners Guide, let us get directly to the heart of the issue by examining the `plot` options using `helpoptions`:
In a nutshell, the important options are ones for whether the screen is erased, for integer axis labels, for logarithmic axes, for even whether the set of axes are drawn, for forcing the axes limits, for plotting one expression versus another, and for forcing the axes to pass through an arbitrary (x,y) coordinate. Of course, as with all define options, these can be used in any combinations.

To simplify the entry of the expressions used in many of the plot examples below, we now declare two functions, \( yv \) and \( xv \), of a single parameter \( t \). Later you will see that they represent a parameterized curve in two-space. Function \( yv \) is a sine wave with a growing amplitude while \( xv \) is a cosine wave with the same amplitude growth but a slightly different period:

\[
1> \text{function } yv(t) \\
1> \quad \text{return } (t^*0.95 \ast \sin(t)) \\
1> \text{function } xv(t) \\
1> \quad \text{return } (t^*0.95 \ast \cos(1.05\ast t))
\]

First of all, let's set the viewport for the entire screen and see what the two waveforms look like. (Be sure to first set the terminal type using, e.g., \textit{terminal hp2648a}):
The graphical output from this command is shown at the top of figure 4.1 for the HP2848a terminal. Before proceeding, there is a system file that contains four defines for setting the four viewports that we shall repeatedly use:

```
list <vport4>

## vport4 - Commands "vport1" through "vport4" for entering four viewports.
define (vport1,viewport 0 0 .5 .5)
define (vport2,viewport .5 0 1 .5)
define (vport3,viewport .5 .5 1 1)
define (vport4,viewport 0 .5 .5 1)
```

These viewports are in the order lower left, lower right, upper right, and upper left, i.e., counterclockwise starting in the lower left quadrant. Now we are ready to begin the demonstration.

```
vport1
plot yv(t) xv(t) vs t from 0 to 12 by .1
```

To avoid having the screen erased by each of the following plot commands, we can turn off the ~erase option on the plot command. (Alternatively, we could just type ~/erase on every plot command.) We then proceed with demonstrating various plot options:

```
set option plot ~erase=NO
```

These four plots are shown at the bottom of figure 4.1. To continue trying more options, try the following:
Figure 4.1. Output From Various Plot Commands.
In viewport 2 (from vport2) we’ve just plotted function $y_v(t)$ versus function $x_v(t)$, over independent parameter $t$. In viewport 3 we force the axes to pass through the origin (coordinate 0,0) and in viewport 4 we add integer axis labels for neatness. These four plots are shown at the top of figure 4.2.

The following plots are shown at the bottom of figure 4.2:

In viewports 1 and 2, we specified the coordinate(s) through which the axes pass for the ~origin option. Viewport 3 demonstrates doing a plot in which the Compiling plot loop message is turned off and the axes are drawn first (by default, they are drawn after the y-expression curves). Finally, viewport 4 shows that the ~origin option can be used without the ~vsexpr option.

5 Additional I/O Features

This section introduces several I/O features that were not discussed in the Beginners Guide. Section 5.1 demonstrates how users can write interactive programs using procedure answer_to_prompt. Section 5.2 addresses the subject of input and output to and from files. In particular, section 5.2.1 is directed towards the pesky openhdlt file and the <FILENAME> convention. In Section 5.2.2 we explain various built-in functions for opening and outputting to files. Sections 5.2.3 and 5.2.4 introduce and apply the special built-in function opuniq for opening unique temporary (scratch) files.
Figure 4.2. More Output From Plot Commands.
5.1 Writing Interactive Programs With Answer_to_prompt

In many situations, it is convenient to prompt the user for input to be entered directly from the terminal. In the Beginners Guide, how to read input from the terminal using readf was demonstrated. For prompting the terminal with a message or request such as Enter number of items, there is command prompt which automatically flushes the output and insures that the next input comes from the terminal and not from pushback (see section 6.1). Although a more powerful approach to prompting the user is the subject of this section, let us first show how to use prompt and readf for this purpose:

```
1> procedure testprompt 
1  |   prompt 'Enter any number: ' 
1  |   readf ' \n* x 
1  |   printf 'Number read = \n* x 
1  | 
1> testprompt() 
Enter any number: 5.3 
Number read = 5.300 
1>
```

Procedure answer_to_prompt is a much more powerful way of prompting the user for input and collecting one or more answers. The procedure is contained in a file called <anspromp> which must be included before using the procedure since it is not automatically built into DELIGHT. Let's illustrate how one may use answer_to_prompt with a simple example:

```
1> use <anspromp> 
1> array name(ARRAYSIZEX) 
1> answer_to_prompt (Enter name: \n, ABB, ABB, GET_NAME, name) 
Enter name: GB? 
(ILLEGAL ANSWER) Enter name: ** 
(ILLEGAL ANSWER) Enter name: R2D2 
1> printf 'Name is \n* name 
Name is R2D2 
1>
```

The prompt in quotes is a string that may, in fact, contain up to two printf-like "%" field descriptors that are associated with the next two procedure arguments. In the above, these arguments are passed as ARB to indicate that they are arbitrary, i.e., just placeholders that are not used since there are no "%" descriptors in the prompt string. GET_NAME is the mode argument (of several possible modes) that requests answer_to_prompt to accept as answer only a name token (see section 6.1), i.e., only a sequence of letters or digits beginning with a letter; as shown above, the prompt is repeated until a valid name is entered. The valid name is returned in the array name, as shown by the printf statement above.

There are many other modes besides GET_NAME. For example, answer_to_prompt can return a single letter answer by using mode GET_LETTER as shown below:

```
1> use <anspromp> 
1> array name(ARRAYSIZEX) 
1> answer_to_prompt (Enter name: \n, ABB, ABB, GET_LETTER, name) 
Enter name: GB? 
(ILLEGAL ANSWER) Enter name: ** 
(ILLEGAL ANSWER) Enter name: R2D2 
1> printf 'Name is \n* name 
Name is R2D2 
1>
```
The **GET_NAME** mode allows you to input only the letters a through z or A through Z. For mode **GET_NAME**, the function value returned by `answer_to_prompt` also contains the single character of the answer:

```plaintext
> array letter(2)
> answer_to_prompt('Sex (M=male, F=female): ', ABB, ABB,
>     GET_LETTER, letter)
Sex (M=male, F=female): 2
(ILLEGAL ANSWER) Sex (M=male, F=female): M
> printf 'Sex is %s' letter
Sex is M
>
```

In fact, all of the various modes return something as the function value and some character string in the last procedure argument (`letter` above). What is returned in each of these is shown in the table at the end of this section.

To see how to use "%" fields in the prompt string and their associated arguments, try the following, which fills a two-by-two array:

```plaintext
> array number_str(MAXTCLSIZE)
> array A(2,2)
> for i = 1 to 2
>     for j = 1 to 2
>         value = answer_to_prompt('Enter A(%i,%i): ', i, j,
>             GET_INTEGER, number_str)
>         A(i, j) = value
>     
> Enter A(1,1): 1.1
> Enter A(1,2): 1.2
> Enter A(2,1): 2.1
> Enter A(2,2): 2.2
> printf A
Matrix A(2,2):
    1.1 1.2
    2.1 2.2
>
```

In this section we have covered the three modes **GET_NAME**, **GET_LETTER** and **GET_NUMBER**. All mode arguments available for `answer_to_prompt` are listed in the following table:
### Procedure `answer_to_prompt Modes`

<table>
<thead>
<tr>
<th>Mode</th>
<th>Allowable User Answer</th>
<th>Function Value Returns</th>
<th>Last Arg Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_DIGIT</td>
<td>Any digit from 0 to 9</td>
<td>The character digit</td>
<td>String consisting of just the character digit</td>
</tr>
<tr>
<td>GET_LETTER</td>
<td>Any letter a-z or A-Z</td>
<td>The character letter</td>
<td>String consisting of just the character letter</td>
</tr>
<tr>
<td>GET_LLLETTER</td>
<td>Any letter a-z or A-Z</td>
<td>The character letter,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>after converting it to lower case</td>
<td></td>
<td>String consisting of just the lower case letter</td>
</tr>
<tr>
<td>GET_NAME</td>
<td>Any sequence of letters or digits beginning with a letter</td>
<td>Length of name</td>
<td>String containing the name</td>
</tr>
<tr>
<td>GET_NUMBER</td>
<td>Any real or integer number</td>
<td>The numeric value of</td>
<td>String containing the number</td>
</tr>
<tr>
<td>GET_NUMBERS</td>
<td>Any sequence of real or integer numbers up to the first non-number token</td>
<td>The number of</td>
<td>Array containing the numbers (not a string)</td>
</tr>
<tr>
<td>GET_STRING</td>
<td>Absolutely anything up to the first blank or to the end of the line</td>
<td>Length of string</td>
<td>String containing the answer string</td>
</tr>
<tr>
<td>GET_EXPRESSION</td>
<td>Any Rattle expression (ending at the first blank or tab following balanced parenthesis)</td>
<td>Length of expression string returned in the last argument</td>
<td>String containing the expression</td>
</tr>
<tr>
<td>GET_ANYTOKEN</td>
<td>Absolutely any input token including NEW-LINE</td>
<td>Length of token string returned in the last argument</td>
<td>String containing the token</td>
</tr>
</tbody>
</table>

There is one final feature of procedure `answer_to_prompt`: if a colon (";") alone is given as an answer, Rattle execution is immediately suspended, just as if a hard interrupt had been generated. Typing `resume` gets you executing inside `answer_to_prompt` again, which first reprompts you with the prompt string argument:

```rattle
1> procedure testprompt {  
  import number_str  
  1| value = answer_to_prompt ('Enter any number: ',  
  1|   ABB, ABB, GET_NUMBER, number_str)  
  1| printf 'Number read = %r/a' value  
1}  
1> testprompt()  
Enter any number: :  
>>> Type "resume" to continue <<<  
2> print sin(5.3)  
-.8323
```
5.2 File Input and Output

The four subsections in this section show how to use available built-in functions for doing file I/O (input/output).

5.2.1 The Openhdtl File

A very useful feature in any system which deals with files is to be able to open and read files which are not in your current directory\(^1\) and which may not even be your own! The convention adopted in DELIGHT is that any filename surrounded by triangular brackets ("<" and ">") such as <graphics> does not exist in the current directory but instead, exists in one of several "standard places" in the operating system file structure, i.e., it exists in another directory. The standard places are specified in the file openhdtl, the subject of this subsection.

The purpose of the openhdtl file (standing for "open-head-tail"), is to locate files whose names are surrounded by triangular brackets. If file openhdtl exists in the directory in which DELIGHT is being run, it is used. Otherwise, a standard openhdtl file is used.

This file consists of pairs of lines containing head and tail strings which are appended before and after filenames surrounded by the brackets. Each head/tail pair corresponds to one standard place in which to look for the file. The first place tried is using the filename obtained by appending the first openhdtl head-string before the specified file name and the first openhdtl tail-string after. If the file does not exist in that location, i.e., this appended filename can not be opened, then the second head/tail pair is tried, and so on. Note that if either the head or tail string is to be empty, then a blank line must be left in file openhdtl. All the pairs of lines in file openhdtl are read once—when the first filename surrounded by triangular brackets is encountered internally by DELIGHT file-opening routine openp, discussed in the next section.

The following example of file openhdtl is for UNIX, in which files in different directories can be accessed by preceding the filename with the directory name, i.e., the tail strings are all null. Identifying comments are shown in parenthesis and are not part of the file:

```
/usr/optcad/nye/include/ (head 1)
/ (tail 1)
/usr/local/lib/ (head 2)
/ (tail 2)
/share1/helper_files/ (head 3)
/ (tail 3)
/share2/helper_files/ (head 4)
/ (tail 4)
```

Based on what has been said above, to open, say, file <mouse>, DELIGHT would try to

\(^1\) By directory we mean the group of files with which you can work with and have control over directly by specifying their unappended filenames.
open the following files (and in the order shown) until one was found:

```
/usr/optcad/nye/include/mouse
/usr/local/lib/mouse
/share1/helper_files/mouse
/share2/helper_files/mouse
```

On some computer systems, however, the head strings might be null with the directory specified in the tail string as in `filename:directoryname`. In this case file `openhdl` might appear:

```
:share1 (head 1)
:share1 (tail 1)
:share2 (head 2)
:local:lib (head 3)
```

and to open file `<mouse>`, DELIGHT would try to open the files

```
mouse:share1
mouse:share2
mouse:local:lib
```

Usually, the first couple of pairs in file `openhdl` are the standard places for the DELIGHT library files. After these pairs, users may add their own head/tail pairs in order to share files from common standard places. Currently, a maximum of 20 pairs are allowed in file `openhdl`.

**`<FILENAME>` Convention.** What happens if the same file exists in directories from two or more different `openhdl` head/tail pairs, for example, if, in the above UNIX example, both of the files

```
/usr/optcad/nye/include/mouse
/usr/local/lib/mouse
```

exist, and you want to get at the second one using only the filename `mouse` and a little extra hint to use the second head/tail pair? To do this there is a simple extension to the `<FILENAME>` convention that allows you to choose which head/tail pair will be used. It is called the **`<FILENAME>` convention** since a pattern is specified inside the triangular brackets before the filename such as pattern `lib` in `<lib/mouse>`. The pattern is first searched for on the head/tail pairs lines. If the pattern is found in a head or tail string, that pair is used to try to open the file, i.e., that pair is appended to the filename as shown above.

Note that it is important not to introduce any machine-dependencies by use of this feature. For example, suppose on UNIX that your `openhdl` file contained the pair

```
/usr/optcad/nye/include/ (blank line)
```

Rattle procedures that opened, say, file `<include/graphics>`, might not run on another computer since the pattern `include` might not exist in any of the head/tail pairs in the other computer's `openhdl` file. For this reason, there is a way of placing characters on a head or tail line that are not actually part of the head or tail string, but whose purpose is simply to match the specified pattern specified in `<FILENAME>` convention. These characters are placed after a backslash ("\"), which in turn is placed after the head or tail string on the same line as in
Then, if on another computer the files in this directory get placed into a directory with tail string :local:lib, the openhdl lines would have to be

:local:lib\include

so that the string include in the filename <\include/graphics> would be found on this openhdl line. Note that file <graphics> would still be found in either file /usr/optcad/nye/include/graphics or graphics:local:lib for these hypothetical examples.

Modifying The Openhdl File. To modify the openhdl file, assuming you don't already have one among your files in the directory in which you are running DELIGHT, you could simply use the DELIGHT built-in editor by typing edit <openhdtl>, make any changes desired, then write out file openhdl in your own directory using w openhdl. Since DELIGHT only reads this file once—and it has already been read when DELIGHT was started—there needs to be a command to tell DELIGHT that you want it to reread file openhdl.

Reset_openhdl Command. The command "reset_openhdl" resets an internal DELIGHT flag that is set after the openhdl file has been read. Thus, if you modify the openhdl file, you must then type reset_openhdl so that the next attempt to open a file with filename surrounded by triangular brackets (such as <graphics>) causes the openhdl file to be reread.

5.2.2 File I/O With Built-in Functions

In Rattle programs, input and output (I/O) may be performed to and from files as well as to and from the terminal. To perform I/O with a file, the so-called present input or present output—where DELIGHT is currently reading input from or writing output to—must be switched to a logical unit number which has been opened to the file. Usually this logical unit number is returned as function value of built-in function openp and is then passed as an argument to built-in routine sochan ("set-output-channel") causing all subsequent output to go into the file opened by openp. After outputting to the file, the present output is restored to what is was previously—in the case of the example below, to the terminal—by a call to built-in routine rochan ("reset-output-channel"). If no further I/O to the file is required, the file can be "closed" by passing its logical unit number to built-in routine close. All of this is demonstrated in the following example:

```
1> unit_mm = openp('myfile',CREATEFILE)
1> sochan(unit_mm)
1> printf 'This should be in file.
1> rochan()
1> close(unit_mm)
1> list myfile
------------------------- Begin myfile -------------------------
This should be in file.
------------------------- End myfile -------------------------
1>
```

When the file myfile was opened above, it was opened with mode CREATEFILE, which
means to create the file if it does not exist. If it does exist, it is simply opened, ready to be written over. Alternatively, if the file exists or you do not want to create the file if it does not, the mode can be passed as WRITEMODE, which opens the file ready to be appended to. With this mode, if the file does not exist, ERROR (defined to be -1) is returned as the function value by openp. Try the following:

```lisp
1> unit_mm = openp('myfile', WRITEMODE) 
1> sochan (unit_mm) 
1> printf 'Should be another line./n' 
1> rochan() 
1> close(unit_mm) 
1> list myfile 

-------------- Begin myfile -------------------
This should be in file.
Should be another line.
-------------- End myfile ------------------- 
1>
1> unit_mm = openp('DoesNotExist', WRITEMODE) 
1> print unit_mm 
-1.000 
1>
```

Finally, if you just want to read from an existing file, READMODE can be passed to openp. As before, ERROR is returned as the function value by openp if the file does not exist. For reading, built-in routines sochan ("set-input-channel") and rochan ("reset-input-channel") are for dealing with the present input analogous to sochan/rochan for present output. So we can read a string from the first line of file myfile using readf (see DELIGHT For Beginners section 7) as follows:

```lisp
1> array string(MAXTOKS/23) 
1> unit_mm = openp('myfile', READMODE) 
1> !
1> sochan (unit_mm) 
1> readf 't*5/n' string 
1> rochan() 
1> !
1> printf 'String = "t*5/n" string 
String = "This" 
1>
```

A very important word of caution: if the three indented statements above had not been in curly brackets, the sochan(unit_mm) statement would have immediately switched the present input to file myfile causing DELIGHT to start reading commands from that file up to the EOF (end-of-file), in which case DELIGHT would get hung. Since a hard interrupt somehow ends the state of being hung, you can try this. First, however you must rewind the file so that reading will begin at its beginning:
Finally, it is a good idea to close all files that are opened as soon as you don't need to use them any longer:

```
1> close (unit_mmm)
```

### 5.2.3 Opening Temporary Scratch Files With Opuniq

Besides the obvious need to open files using `openp`, there is quite often a need to create a temporary file, only to exist a short while, that is later eliminated. A Rattle programmer developing commands and procedures for other DELIGHT users could easily just create a file with name, say, `temp` in the user's directory. But this raises two serious questions. First, how does the procedure developer know that he is not using the name of and thus overwriting an existing file in the user's directory? Second, what names should be used if several scratch files are needed simultaneously? Clearly, using filenames like `temps1`, `temps2`, etc., only increases the chance of coinciding with an existing user filename.

A rather elegant approach is to have DELIGHT generate the scratch file in another directory—thus avoiding filename conflicts—and also guarantee that the filename is unique, thereby allowing several scratch files to be open at the same time. This service is provided in DELIGHT by built-in function `opuniq`, which is used according to the syntax:

```
unit_mmm = opuniq (NAME_FRAGMENT, ACTUAL_NAME_USED, MODE)
```

This function, just like `openp` of the previous section, returns as function value a logical unit number that can be passed to `sochan` to direct subsequent output to this file. It opens a unique, temporary file having a filename that uses, if possible, the characters in string `NAME_FRAGMENT`, returning the actual filename used in string `ACTUAL_NAME_USED`. The open mode is specified in argument `MODE` and is passed directly to the last argument of function `openp`.

To demonstrate how you may use `opuniq`, an example template already exists in a file called `<Topuniq>` in a standard directory of the DELIGHT system. All you need to do is edit the file and modify it. First, let's list the file:

---

1 If you have a `openhdtl` file (see section 5.2.1), it must contain an entry that can find file `<Topuniq>`. If the following `list` command returns with `ERROR: list: Cannot open "<Topuniq>"`, see system personnel to update your `openhdtl` file or simple rename it to another name besides `openhdtl`.**
32 DELIGHT For Intermediates (3/1/85)

1> list ~numbers <Topuniq>
1> 1## This example creates a temporary file, prints some commands into
1> 2## it, puts a final command into it to exit and remove itself, then
1> 3## pushes back an include statement for the file.
1> 4array actual_name(40)
1> 5unit_num = opuniq('tempfile', actual_name, CREATEFILE)
1> 6printf 'ERROR: command: Cannot open or create temporary file
1> 7printf "%p"/n' actual_name
1> 8suspend
1> 9 sochan (unit_num)  ## Set present output channel.
10# . (printf ...)  ## Print commands into the file.
11# .
12printf 'exit; filpm("%p")/n' actual_name
13rochan ()  ## Reset present output channel.
14pbf 'include %p/n' actual_name  ## Push back the include statement.

The `pbf` statement above (discussed in detail in section 8.1) pushes back an `include` statement for the actual filename opened by `opuniq`, causing that statement to be the next input read by DELIGHT. The `%p` is for a packed string and follows the same conventions as the "%" format control fields discussed in the Beginners Guide for `printf`.

In order to make this example really useful, we need to replace the three lines (13th through 15th) with some real commands:

1> edit <Topuniq>
1> "<Topuniq>" 18 lines
1> :13,15p
1> # . (printf ...)  ## Print commands into the file.
1> # .
1> :13,15c
1> 3 lines changed
1>   printf 'define (aa,5)/n'
1>   printf 'print aa/n'
1> :w junk
1> "junk" 19 lines
1> :q
1> use junk
1> 5.000
1>

The `5.000` seen after `use junk` is due to writing `print aa` into the temporary file, and then including the file (caused by the `pbf` line above). Other applications of `opuniq` include the creation of temporary files for use in including several other files. This will be taken up further in the next section.
5.2.4 An Application of Opuniq

This section presents a very useful application of function \texttt{opuniq}. A temporary file is created by \texttt{opuniq} and \texttt{include} statements are written into it so that including the temporary file includes several arbitrary files. A Rattle procedure to achieve this is already set up for you in file \texttt{<incfiles>}. List the file as follows, and we shall explain it step by step. See the footnote in the previous section if the \texttt{list} command produces an error message:

```
1> list <incfiles>
---------------------- Begin <incfiles> ----------------------
## incfiles - Implementation of "include_files" command.
define include_files "f1="; "f2="; "f3="; "f4="; "f5="; "f6="
    include_files_(f1,f2,f3,f4,f5,f6)
end

# Help file input.
% include_files Include from 1 to 6 files with a single command.
% include_files FILE1 [ FILE2 FILES ... FILES ]
% include_files <plot> <plot3d> myplot
%A include, use, include_and_print
%SF <incfiles>
%A Bill Nye
%K include file io

procedure include_files_ (f1, f2, f3, f4, f5, f6) |
    array actual_name (40) |
    unit_num = opuniq ('incfiles', actual_name, CREATEFILE)
    if ( unit_num = ERROR ) |
        printf 'ERROR: include_files: Cannot create file "5$>".
    suspend
    sochan (unit_num)
    if ( !eqsp(f1,"")) printf 'include %p/n' f1 |
    if ( !eqsp(f2,"")) printf 'include %p/n' f2 |
    if ( !eqsp(f3,"")) printf 'include %p/n' f3 |
    if ( !eqsp(f4,"")) printf 'include %p/n' f4 |
    if ( !eqsp(f5,"")) printf 'include %p/n' f5 |
    if ( !eqsp(f6,"")) printf 'include %p/n' f6 |
    printf 'exit ; filprm("%p")/n' actual_name |
    rocchan () |
    close (unit_num) |
    pbf 'include %p/n' actual_name |
    array actual_name(0) |
    End <incfiles> ----------------------
```

The define statement creates a new command called \texttt{include_files} which has the six arguments \texttt{f1} through \texttt{f6} that represent the one to six files to be included with one \texttt{include_files} command. Following the defines statement are lines starting with \texttt{%} which are the raw input to the DELIGHT online help system. See [6] for how such help entry lines are set up. Procedure \texttt{include_files—}which demonstrates the convention explained in section 3.5 of making the procedure name the command name plus an underscore—executes when the \texttt{include_files} command is issued. The body of the procedure is almost identical to the example of the previous section except for the \texttt{printf} statements that write the \texttt{include} statements into the temporary file. To show that
this program can be used to include several files with just one command, we create three dummy files and use the command as follow:

```
1> edit junk1
Unable to open "junk1"
:a
printf 'Inside junk1/n'
:w
"junk1" 1 lines
:e junk2
Unable to open "junk2"
:a
printf 'Inside junk2/n'
:w
"junk2" 1 lines
:e junk3
Unable to open "junk3"
:a
printf 'Inside junk3/n'
:wq
"junk3" 1 lines
1> list junk3
------------- Begin junk3 ----------------
printf 'Inside junk3/n'
------------- End junk3 ---------------
1> use <inofiles>
1> include_files junk1 junk2 junk3
inside junk1
inside junk2
inside junk3
1>
```

As can be seen, the three dummy files have been included with just one `include_files` command. You can turn on the echoing of DELIGHT input lines, if desired, to help you debug a procedure that uses a temporary file:

```
1> echo
echo
1> linc
>> linc
include_files junk1 junk2 junk3
>> include junk1
>> printf 'Inside junk1/n'
inside junk1
>> include junk2
>> printf 'Inside junk2/n'
inside junk2
>> include junk3
>> printf 'Inside junk3/n'
inside junk3
>> exit ; filprm("/tmp/inofilesA29305")
1> noecho
>> noecho
1>
```

The scratch filename shown as argument to built-in procedure `filprm` (for removing files) is from running this example and creating the temporary file on the UNIX system. (Even if you are running on UNIX, however, the trailing digits in the scratch filename are
DELIGHT's process id and will surely be different from those above.)

6 Language Extensibility Using Macros

DELIGHT has extensibility needs that cannot be handled by the simple define substitution mechanisms discussed in section 3 and in the Beginners Guide. These have to do with making conditional substitutions, that is, substitutions that are based on the arguments that are used with the define. For example, there is no way to make a define called MatrixFunc which allows the statement MatrixFunc A=inv(B) to substitute the procedure call Inverse(A,B) but the statement MatrixFunc A=adj(B) to substitute Adjugate(A,B). The definition substituted when a define is encountered in input source is fixed in structure; only arbitrary argument values can be substituted into the appropriate places in the definition. Section 6.1 introduces the concepts of tokens and the push-back mechanism which are important in solving the above substitution problem using Rattle macros, presented in section 6.2.

6.1 Tokens and Push-Back

Before we undertake one of the most important features of DELIGHT, namely, macros, it is essential that readers have some understanding of the Rattle compiler, the push-back mechanism, and how they relate to a Rattle source program. In this section, we shall discuss these basic concepts. Although all readers are encouraged to read it, those familiar with compilers may wish to proceed directly to section 6.2.

A compiler takes as input a source program and produces as output an equivalent sequence of instructions [1]. This process is so complex that it is not reasonable, either from a logical point of view or from an implementation point of view, to consider the compilation process as occurring in one single step. For this reason, it is customary to partition the compilation process into a series of subprocesses called phases, as shown in figure 6.1. The first phase, called the lexical analyzer or scanner, separates characters of the source language into groups that logically belong together; these groups are called tokens. Thus each token represents a sequence of characters that can be treated as a single logical entity. The usual tokens are keywords, such as while or if, identifiers, such as X, help, or vector, operator symbols such as <= or +, and punctuation symbols such as parentheses or commas. The output of the lexical analyzer is a stream of tokens, which is passed to the next phase, the syntax analyzer, or parser.

As a simple example, let's consider the define define(PI, 3.1416). This define makes PI get substituted by 3.1416 when a command such as print PI is issued. To achieve this, the Rattle compiler must have an internal mechanism for substituting PI with 3.1416. The mechanism for allowing the Rattle compiler to "receive" input (i.e. 3.1416 in this case) which was not actually typed in by the user nor in a file being included is called the push-back mechanism.

We begin our explanation of the push-back mechanism by introducing procedure gtoken and command pbf. Recall that the lexical analysis phase of a compiler reads the source program one character at a time, carving the source program into a sequence of logical units called tokens. The built-in routine gtoken (for "get-token") is one way that this can occur; when called, it returns one token read from the input (or from any characters presently pushed back—see the next paragraph). If the token is a character string, it is returned in the first argument of gtoken; if the token is a number, gtoken's second argument contains the numeric value. The syntax of a call to
Figure 6.1. Compilation Phases.
*gtoken* is thus

\[ gtoken \text{(TokenString, Value)} \]

To try a simple example of how *gtoken* works, try the following:

```plaintext
1> array name_str(MAX_TOKENS128), number_str(MAX_TOKENS128)
1> { prompt 'Enter name and number:
1> gtoken (name_str, dummy)
1> gtoken (number_str, value)
1> Enter name and number: hello 234
1> printf 'name="%s", number="%s", value=%r/n' name_str number_str value
name="hello", number="234", value=2.340e+2
```

This shows that the first argument returns the characters of the input token even if it is a number.

Characters can be pushed back onto the input so that they are the next to be read by DELIGHT. This read can either be internal to DELIGHT in its reading of commands from the terminal, or by Rattle execution of any means of reading input such as *gtoken*, *readf*, etc. (all DELIGHT input follows the same push-back mechanism). The push back is performed using the *pbf* statement, having syntax

\[ \text{pbf 'CONTROL-STRING' [ ARG1 ARG2 ... ARG6 ]} \]

When a *pbf* statement is executed, it pushes back onto the input a formatted control string with optional 0 to 6 arguments with exactly the same control string meaning as for *printf*. Let's use *pbf* to push back a Rattle statement which is then immediately read by DELIGHT and executed:

```plaintext
1> pbf 'print 1/n'
1.000
1> k = 5
1> pbf 'print %d/n' k
5.000
```

To understand conceptually how pushback works, consider the following table in which the left column contains Rattle code executed, the middle column contains a comment concerning the effects of the execution, and the right column shows characters remaining to be read by the Rattle lexical analyzer, that were either typed in by the user or previously pushed back. Here, as for the *printf* statement, \( /n \) indicates a NEWLINE character, that is, the fictitious character at the end of an input line.
This example is actually how the definition for the define \textit{PI} would be substituted internally for the occurrence of the define name \textit{PI}. This internal substitution process is entirely invisible to the user, that is, all the get-token and push-back functions involved in the above table need not be programmed by the user but are automatically executed when a define name is encountered. However, the next section introduces macros, in which you are required to program using \textit{gtoken} and \textit{pbf}.

### Rattle Pushback Mechanism

<table>
<thead>
<tr>
<th>Rattle Code Executed</th>
<th>Effect of Execution</th>
<th>Remaining Input Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtoken(TokenString, Value)</td>
<td>TokenString now contains &quot;1&quot; and Value equals 1.</td>
<td>1+PI/n</td>
</tr>
<tr>
<td>gtoken(TokenString, Value)</td>
<td>TokenString now contains &quot;+&quot;.</td>
<td>+PI/n</td>
</tr>
<tr>
<td>gtoken(TokenString, Value)</td>
<td>TokenString now contains &quot;PI&quot;.</td>
<td>PI/n</td>
</tr>
<tr>
<td>pbf '3.1416'</td>
<td>Push back the character string &quot;3.1416&quot;.</td>
<td>/n</td>
</tr>
<tr>
<td>gtoken(TokenString, Value)</td>
<td>TokenString now contains &quot;3.1416&quot; and Value equals 3.1416.</td>
<td>3.1416/n</td>
</tr>
<tr>
<td>gtoken(TokenString, Value)</td>
<td>TokenString now contains &quot;/n&quot;. Now, the next line of input would be read in.</td>
<td>/n</td>
</tr>
</tbody>
</table>

6.2 Language Extensibility Using Macros

As mentioned in the introduction to section 5, the DELIGHT extensibility needs that cannot be handled by the define substitution mechanism have to do with making conditional substitutions that are based on the arguments that are used with the define. The way DELIGHT provides this language extensibility is through the procedure-like entity called a Rattle \textit{macro}. Macros are written in the Rattle language in exactly the same way as procedures, except the keyword \textit{procedure} is replaced by \textit{macro} and they do not have an argument list surrounded by parenthesis. Also, they do not execute at runtime (as procedures do), but rather when their name is encountered by the lexical analysis phase during the compilation of Rattle statements. Macros can act as preprocessors that modify the input character stream being passed through the lexical analyzer on to the Rattle parser. For example, one can write a macro to scan the next few input tokens—which may not even be valid Rattle code since they never reach the parser but are only "seen" by the macro—make decisions based on what is found, and then push back valid Rattle code that eventually is passed to the parser. Hence the general process undertaken by a macro is:
1. Get the next few tokens,
2. Make decisions based upon these tokens, and
3. Push back substituted text that is usually valid Rattle code.

Let us now implement the simplest possible macro, in fact, so simple that it does not even read any tokens or make decisions but just pushes back a single number token:

```
1> macro Five
1| pbf '5.0'
1> print Five
5.000
1>
```

Now let's put a print statement inside the macro so that we can see when the macro actually executes:

```
1> macro Five {
1|    printf 'Inside macro/n1
1|    pbf '5.0'
1| }
1> print Five
Inside macro
5.000
1>
```

To show that the macro actually executes when the statement using the macro is compiled — when the macro name is detected — and not when the statement itself executes, we place curly brackets around our print statement to prevent it from executing until the closing bracket is given:

```
1> { 
1|    print Five Five
Inside macro
Inside macro
1| 5.000 5.000
1>
```

As you can see the macro `Five`, by virtue of where you see the output `Inside macro`, has executed before the `print` statement containing it executes.

We are now ready to implement a macro that does read input tokens and use them to make decisions about what to push back. We will implement the `MatrixFunc` macro mentioned in the introduction to section 6. The task of this macro is to read input tokens that make up the `MatrixFunc` statement and, depending on the which of the keywords, `inv` or `adj` is read, push back the appropriate procedure call, either `inverse` or `adjugate`. In other words, the macro is to convert

```
MatrixFunc A=inv(B) into Inverse(A,B)
```

and
Since this macro is a little too long to have you type in, there is a file containing it, listed below; all you have to do is use (include) it:

```plaintext
macro MatrixFunc
    array OutputMatrix(MAXTOKSIZE), InputMatrix(MAXTOKSIZE),
    Function(MAXTOKSIZE), Dummy(MAXTOKSIZE)
    define (EqualString, eqstp)  # Logical function for string comparison.
    gtoken (OutputMatrix, Value)  # Read "A".
    gtoken (Dummy, Value)  # Discard ".
    gtoken (Function, Value)  # Read function.
    gtoken (Dummy, Value)  # Discard "(".
    gtoken (InputMatrix, Value)  # Read "B".
    gtoken (Dummy, Value)  # Discard ")".
    if ( EqualString (Function, 'inv') )
        pbf 'Inverse(%s,%s)/n' OutputMatrix InputMatrix
    else if ( EqualString (Function, 'adj') )
        pbf 'Adjugate(%s,%s)/n' OutputMatrix InputMatrix
    else
        printf 'ERROR: MatrixFunc: Illegal function: "%s"/n' Function
    
procedure Inverse (a,b)
    printf 'Inside procedure Inverse./n'
procedure Adjugate (a,b)
    printf 'Inside procedure Adjugate./n'
```

In file `<Tmatfunc>` above, the various arrays declared are each for holding a character string containing an input token returned as the first argument of routine `gtoken` (see section 6.1). When `MatrixFunc` is used above, the detection of the macro name by `DEUGHT` causes `DEUGHT` to execute it immediately. Thus, `gtoken` reads the next few items from the input, namely, tokens `A = inv ( B and )`. Then the macro pushes back one of the two procedure calls, `Inverse` or `Adjugate`, based on which of the keywords `inv` or `adj` was read by the macro.

That the `DELIGHT` macro feature presented in this section is certainly powerful is substantiated by the fact that all of the matrix macros in section 13 of `DELIGHT For Beginners` have been created using the same tools and techniques shown in this section.

### 7 Debugging Rattle Programs

This section looks at several ways of debugging both compiler-reported errors such as syntax errors as well as run-time errors. Debugging compiler errors is discussed in section 7.1 where we introduce ways of tracing what various macros push back. Section 7.2.1 through 7.2.4 clarify what is meant by run-time errors and how you use
available commands such as trace, enter, display local variables, and suspend to locate where the problem occurs. Section 7.2.5 demonstrates how you make DELIGHT abort immediately upon an overflow. This is useful when you're trying to determine exactly where an executing built-in routine overflows, assuming your operating system has a way of revealing this information through messages on the screen or some kind of program debugger. When in really deep trouble in DELIGHT, as a last resort, the command hardreset, covered in 7.2.6, can be used to reset several internal states, buffers, etc., in DELIGHT. Section 7.3 then discuss how to use the whatis and whereis commands for debugging and, in general, for learning about details of how things have been implemented in DELIGHT.

7.1 Debugging Compiler-Reported Errors

The Rattle parser reports syntax errors by printing the offending input line, pointing to the approximate location of the error with a caret ("~"), and giving an error message. For example:

```
1> print 1/3
print 1/3
```

```
  ERROR(1) Expression syntax error [print 1/3 ]
```

The 10 characters in square brackets are, approximately (as can be seen), the last 10 characters read from pushback by an internal DELIGHT function that returns input characters. Showing these sometimes helps in finding an error in a define definition as shown:

```
1> define (PI,4*atan(1))
1> print sin(PI)
```

```
  ERROR(1) Expression syntax error [4*atan 1 ]
```

```
1> define (PI,4*atan(1))
1> print PI
3.142
```

Whether the 10 characters in square brackets above help you find the error in the definition of PI is unclear, but their being printed along with the error message is probably better than just having the caret symbol point at the input sin(PI), which "looks" reasonable.

The following subsections are directed toward debugging errors reported by the Rattle parser. In particular, section 7.1.1 shows how one can trace what various macros push back by setting the system variables trace_pushback_ and trace_matep_. Other debugging suggestions are given in section 7.1.2.
7.1.1 Tracing What is Pushed Back

Many commands in DELIGHT are macros that push back Rattle code as demonstrated in section 5. In many of these, there is a call to a built-in routine called pbdump (for "pushback dump") which prints to the screen everything that is presently pushed back, i.e., that prints out the entire contents of the pushback stack. Furthermore, the call to pbdump usually occurs only if system variable trace_pushback has been set to YES (as opposed to NO, both Rattle defines). The output of pbdump is sometimes useful in debugging certain types of errors reported by the parser. For example, suppose you (accidentally or otherwise) pass an expression to printv instead of just a variable or array name:

```
1> a = 2
1> printv a/3
Scalar a = 2.00000
ERROR: Illegal statement: "/"
1>
```

The error message above is not very clear. However, this situation can be mitigated by setting trace_pushback:

```
1> trace_pushback=YES
1> printv a/3

------ Push-Back-Dump from "printv" ------
printf "Scalar a = %f
/n" a
/
------------------------------------------
Scalar a = 2.00000
ERROR: Illegal statement: "/"
1>
```

From the pushback dump, you can see how, after reading the printf statement, DELIGHT next reads a statement containing just a slash ("/"). To show that this gives the same error message, try the following:

```
1> /
ERROR: [Illegal statement: "/"
1> reset
1>
```

The prompt changing to 1} usually means that the previous statement is incomplete. Here, it has to do with the fact that an expression that ends in an operator is automatically continued onto the next line (see the discussion of expression and assignment continuation in section 4.5 of DELIGHT For Beginners). Although entering another blank line when 1} is seen above will restore the prompt to 1>, just to be safe, reset was typed above.

You should use trace_pushback and pbdump in your own macros to aid in debugging. For example, we can add their use to the trivial macro Five of section 5.2:
The call to `pbdump` shows everything that is currently pushed back. If something else such as a definition body is presently pushed back, it is seen also:

```
1> define(Five3, print Five+3.000000)
1> print Five

----- Push-Back-Dump from "Five" -----
5.0
---------------------------------------
5.000
1>
```

7.1.2 Other Debugging Suggestions

DEIGHT users, over the period of years since DEIGHT has existed, have come across errors in their Rattle code which seem to defy rational explanation! For example, sometimes you get an unexplainable error when compiling a procedure's list of arguments:

```
1> procedure SetArray(xary, clip)
procedure SetArray(xary, clip)
ERROR(1) Name expected
1] reset
reset
ERROR(1) erroneous input token [  ]
1>
```

In such cases, one thing you can do is check that each argument is not a define or something else that already exists in DEIGHT. This can be done by using the `whatis` command, briefly introduced in the Beginners Guide. Its use is as follows:

```
1> whatis xary
"xary" DOES NOT EXIST.
1> whatis clip
"clip" is a macro: From file "<clipmac>".
1>
```
Since the argument clip already exists as a macro, you should change the argument name to something else (such as clipflag).

Another suggestion for debugging is to turn on the echoing of input lines as they are read by DELIGHT. This is important when a procedure exists in a file that is to be included since the lines in the file are then echoed to the screen as they are read and Rattle compiled. Echoing is turned on/off with the commands echo/noecho, already demonstrated in the Beginners Guide.

7.2 Debugging Run-Time Errors

7.2.1 What Run-Time Errors Are

Certain errors which occur during Rattle execution are detected internally by DELIGHT and cause execution to suspend as if a hard interrupt (see section 5 of the Beginners Guide) had occurred. These include:

Floating-point exceptions: These are divide by zero, numerical overflow, bad arguments to built-in Fortran-like functions such as taking the logarithm of a negative number, etc.

Array out-of-bounds: This is when the "net" array subscript for an array goes beyond the total array size or is less than one. For example:

1> array Y(3,2)
1> print Y(1,4)

RUN-TIME ERROR: Array subscript out of bounds: array "Y"
Net array subscript = 10
0.000
Interrupt...
2> reset
1>

The net array subscript is computed, using column-major array subscripting (storage in column order), as

1 + (4-1)*3 = 1 + 9 = 10

and ten is beyond the total array size of 3*2=6.

To clear up the idea of net array subscript, here is an example in which the it is less than the total array size (even though the first subscript is too large) so that the program does not suspend:

1> readmatrix Y
1: 1.1 1.2
2: 2.1 2.2
3: 3.1 3.2
1> printv Y
Matrix Y(3,2):
1.1e+1 1.2e+1
2.1e+1 2.2e+1
3.1e+1 3.2e+1
1> print Y(4,1)
1.200e+1

This statement program did not suspend since the net array subscript is computed as

\[ 4 + (1-1)*3 = 4 + 0 = 4 \]

and 4 is within the total array size of 3*2=6.

7.2.2 Review of Commands for Debugging Run-Time Errors

This section reviews DELIGHT features presented in the Beginners Guide that aid in debugging run-time errors that occur in Rattle procedures. We discuss the `trace` and `enter` commands.

The `trace` command is very useful for debugging DELIGHT run-time errors, as can be seen in the following:

1> edit junkl2
Unable to open "junkl2"
:1c
procedure junkl(x)
  print 1/x
procedure junk2(x)
  junkl (x)
.
:ww
"junkl2" 2 lines
1> use junkl2
1> junk2 (0)
RUN-TIME ERROR: 1 overflow(s) or other floating point exception(s).
0.000
Interrupt...
2> trace
Interrupted IN procedure
  junk1  line 2  of file junkl2
  junk2  line 4  of file junkl2
2> reset
1>

The `trace` output shows that the run-time error occurred in procedure `junk1`, line 2 of the file, which was called by procedure `junk2` on line 4 of the file. Obviously, it is due to the division by x with x equal to zero. Using the editor, you could now examine the source lines that lead to the RUN-TIME ERROR. Alternatively, you could list the file with line numbers:
The `trace` command can be used whenever an interrupt has occurred, i.e., whenever the interrupt level is greater than one thus causing the prompt to appear as `?`, `?`, etc.

Another DEUGHT feature to aid in debugging is the `enter` command for looking at local arrays and variables of a procedure. Let's create a simple procedure with three local variables, `a`, `b`, and `c`:

```
1> procedure junk1 (x)
   a = 1
   b = 2
   c = 3
   print a*x b*x c*x
1> junk1(2)
2.000 4.000 6.000
```

After executing this procedure as above, you may `enter junk1` and look at the local variables:

```
1> enter junk1
e> display local variables *
3 variables:
   a  =  1.00000
   b  =  2.00000
   c  =  3.00000
```

Note that after entering a procedure with the `enter` command, the prompt changes to `?` to remind you that any variables you create or use are actually local to the entered procedure.

### 7.2.3 Using Pdebug For Debugging

This section introduces a system variable called `pdebug`, which prints parse or execution debug concerning built-in routines. To get this debug output, simply set `pdebug` according to the following table:

```
Debugging With pdebug_

<table>
<thead>
<tr>
<th>Value</th>
<th>What is Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No debug output.</td>
</tr>
<tr>
<td>1</td>
<td>Much parser information plus built-in function names and numbers of arguments, right before each function is called.</td>
</tr>
<tr>
<td>2</td>
<td>Just built-in function names and numbers of arguments, and the values of all arguments, right before each function is called.</td>
</tr>
<tr>
<td>3</td>
<td>Same as pdebug_ = 2 except print the values of all arguments right before and right after each function is called.</td>
</tr>
</tbody>
</table>

For example:

```plaintext
1> pdebug_ = 2
1> printf "a=%r b=%r/n" 1.111111 2.222222
>> builtin: ENTERING "printf", funcno=3 nargs=7
   >> builtin: FIRST VALUE OF EACH (REAL) ARG:
   1= 1.378012e-2 2= 1.111111 3= 2.222222 4= 0.000000
   5= 0.000000 6= 0.000000 7= 0.000000
a= 1.111111 b= 2.222222
1>
```

The above shows the arguments to built-in routine printf, whose first argument is the format control string and whose last six are the zero to six arguments in 1-1 correspondence with the % fields of the control string. For more information, type `help pdebug_`.

As a second example, let's rerun procedure LEproc from section 3.2, which we first modify to not print dashes:

```plaintext
1> define (ListEdit "nans.
1> procedure LEproc (none))
1> ListEdit nwf
1> >> builtin: ENTERING "openp", funcno=10 nargs=2
1> >> builtin: FIRST VALUE OF EACH (REAL) ARG:
1> 1= 7.428055e+31 2= 1.000000
1> >> builtin: ENTERING "sichan", funcno=20 nargs=1
1> >> builtin: FIRST VALUE OF EACH (REAL) ARG:
1> 1= 1.200000e+1
```
As can be seen above, the `list` command, the first statement inside procedure `ListEdit`, calls five built-in routines: `openp` to open the file to be listed (see section 5.2.2), `sichan` to set the present input to this file (see section 5.2.2), `cpytoeof` to copy present input to present output up to the end-of-file, `cloze` to close the logical unit number opened by `openp` (see section 5.2.2), and `richan` to reset the present input to what it was before. Finally, the edit statement simply calls built-in routine `exedit`, to invoke the DELIGHT editor. Before going on, you'd better turn off `pdebug_`:

```
1> pdebug_ = 0
```

### 7.2.4 Debugging by Adding Print and Suspend Statements

A technique used heavily by programmers trying to debug a program is to add print statements around the suspected causes of trouble. The ability in Rattle to recompile a procedure without any load/linkage phase is very conducive to such an approach. But an interactive program development environment such as that provided by DELIGHT has another powerful debugging technique—that of placing `suspend` statements around suspected trouble spots. This allows you to enter the procedure, display local variables, `leave`, and resume execution, even after you have modified a local (or nonlocal imported) variable.

We illustrate how adding print and `suspend` statements can aid the debugging effort in the following examples.

```
1> edit junkxy
Unable to open "junkxy"
:a
function xyinv(x, y) {
    return (1/(2*7))
}

."junkxy" 3 lines
```
The overflow caused by the zero denominator halted the execution of procedure \texttt{xyinv}. We now add a print statement to the procedure to help locate the problem. (Note that the source of error for this particular run-time error is quite obvious; a general procedure is being demonstrated here so please do not jump to any hasty conclusions about the authors' intelligence!)

The print statement has helped us to find out that the overflow occurred when \(x = 1000\) and \(y = -1000\) resulted in division by zero. Next we modify procedure \texttt{xyinv} by replacing the print statement by a \texttt{suspend} statement. Entering the procedure and examining its formal argument values—generally not allowed in \texttt{DELIGHT}—is also demonstrated:
7.2.5 Aborting On Numeric Overflow

This section details how to make DELIGHT abort immediately when an overflow occurs. By default, DELIGHT does not abort when a floating point exception such as an overflow or a divide-by-zero occurs; Rattle execution simply suspends with a "RUN-TIME ERROR". Sometimes, however, when, say, debugging a built-in Fortran routine, one would like to abort immediately when the overflow occurs, in order to determine where it occurred. To do this, all you need to do is set the following option:

1> set-option DLE:options ~AbortOnOverflow=YES

At this point we advise against trying a statement now such as print 1/0 since DELIGHT will immediately abort and you will have to restart it. However, let us just look at what might occur (shown below for the UNIX operating system). Instead of a trivial divide-by-zero as print 1/0, let's try causing an internal routine to overflow. We shall make the built-in routine used to multiply matrices overflow by passing two one-by-one matrices each having value MAXREAL, a DELIGHT define for the largest representable floating point number. Then this routine overflows since MAXREAL squared is surely not representable.

1> matop A = array(1) of MAXREAL
1> matop B = array(1) of MAXREAL
1> matop C = A*B
*** Illegal instruction
Illegal instruction (core dumped)
%

DEIGHT has aborted immediately upon the overflow. At this point, to determine where the executing built-in routine overflowed, either you see an abort message on the screen that reveals this or you would have to use some operating system utility such as a program debugger.
7.2.6 DELIGHT Internal Aborts and the Hardreset Command

When serious internal DELIGHT problems occur that are generally impossible to recover from, the infamous ABORT ABORT ABORT ... message appears on the screen and you are asked whether you really want to abort out of DELIGHT or whether you want to return to DELIGHT and "take your chances". One example is when you try to open too many (For UNIX, more than 7) files using openp (see section 5.2.2). Here, we repeatedly open the existing file junkxy that was created in section 7.2.4, noting that the same file can be opened to several logical unit numbers:

```
1> printf 'Xi/n' openp('junkxy',READMODE)
11
1> !!
print openp('junkxy',READMODE)
12
1> !!
print openp('junkxy',READMODE)
13
1> !!
print openp('junkxy',READMODE)
14
1> !!
print openp('junkxy',READMODE)
15
1> !!
print openp('junkxy',READMODE)
16
1> !!
print openp('junkxy',READMODE)
17
1> !!
print openp('junkxy',READMODE)
18
1> !!
print openp('junkxy',READMODE)
19
```

ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT ABORT

* openp: Attempt to open too many files (max=7) *

Create "core" file?
Y.y = Yes, then leave program.
N.n = No core file; just leave program.
R.r = Return to program execution.
Q.q = just like N: immediately Quit program.
? = repeat ABORT message and this prompt.
<CR> = repeat this prompt (max 5 repetitions).

:R

NOTE: All open unit numbers have been closed and are being reused!

11
1> hardreset
1>

The large prompt appearing on the screen after the abort message in the box is machine dependent; the prompt shown above is for UNIX. After returning from an ABORT ABORT ABORT ..., you should give a hardreset unless you know exactly what the implications of returning from the abort are (and believe us, you probably don't!). The hardreset command is used as a last resort to reset all internal states, stacks, and buffers in DELIGHT. In particular, the following are reset or cleared:
- Any characters presently pushed back;
- The input line buffer;
- The output line buffer;
- The input unit number stack set and reset by `sichan/richan`
- The output unit number stack set and reset by `sochan/rochan`
- The unit number being input or output echoed to, if any;
- All open unit numbers opened by `openp`, `opens`, `input_from`, `include`, etc.

For more information, type `help hardreset`.

### 7.3 General Use of Whatis and Whereis

In DELIGHT, all (both built-in and user-created) arrays, defines, functions, macros, operators, procedures and variables are in one large symbol table. The command `whatatis` followed by an entry name shows the type of symbol table entry. The `whereis` command shows the actual filename used—with the head and tail strings from your openhdtl file appended (see section 5.2.1)— for filenames surrounded by triangular brackets such as `<graphics>`.

It can be used to help you find out about a command by allowing you to look at the file containing the actual Rattle source code that implements the command (assuming you have file system permission to read the file). Below are examples of the use of `whatatis` and `whereis`. For the time being, ignore any `@system` that appears on any name:

```
1> whatatis box
"box" is a define: "box@system()".
1> whatatis box_
"box_" is a function or procedure: From file "<graphics>".
1> whereis graphics
File "<graphics>" from "/oe/optcad/nye/include/graphics"
```

The above example shows that `box` is a define with definition `box@system()`, that `box_` is a procedure from file `<graphics>`, and that file `<graphics>` is from the given filename (shown here for UNIX on Esvax at Berkeley). We entered three commands to find out this information about the item `box`. This procedure of issuing two (or more) `whatatis` and one `whereis` commands for a particular item is so frequently used that there is sufficient grounds to have a command called `Whatis` (with a capital "W") which performs the three commands in one:

```
1> Whatis box
"box" is a define: "box@system()".
"box@system" is a function or procedure: From file "<graphics>".
File "<graphics>" from "/oe/optcad/nye/include/graphics"
```

---

1 `@system` has to do with environments, to be discussed in a future version of this document; you are currently in environment `system`. 
What *Whatis* really does is repeatedly call upon *Whatis* as long as the first token of the definition found by the previous *Whatis* is a name (i.e. a sequence of letters or digits beginning with a letter). One small nuisance is that you don't get the *Whereis* call if the last definition does not start with a procedure name, as shown in the following:

1> define(runprint,[algo()];printv X])
1> Whatis runprint
"runprint" is a define: "[algo()];printv x].
1>

*Whatis* has not reported on procedure *algo* since the definition started with character "). It does, however, in the following modification:

1> define(runprint,algo();printv X)
1> Whatis runprint
"runprint" is a define: "algo();printv x."
"algo" is a function or procedure: From file "<Esetup>".
File "<Esetup>" from "/oe/optcad/nye/libmake/Esetup"
1>

This definition is OK if *runprint* is only given as an interactive command. But if it is the single-statement body of an *if* statement as in:

```plaintext
if ( .... )
 runprint
```

then only *algo()* will become the *if* statement body; the *printv X* will be outside, equivalent to:

```plaintext
if ( .... )
 algo()
 printv X
```

Conclusion: don't use defines whose definition consists of two statements unless you are sure that the define will never be used inside Rattle procedures. If it might be used in this way, keep the definition surrounded by curly braces as in either of the following:

```plaintext
define(runprint,[algo()];printv X])

define runprint
 { algo()
  printv X
 }
```

The *Whatis* command also shows the positions of define arguments and define options in the definition string. Arguments are shown with #1 for the first argument, #2 for the second, etc., and &1 for the first option, &2 for the second, etc. For example, the define for the *printfancy* command from section 3.5, which was

```plaintext
define (printfancy ~stars=YES &line=NO X,printfancy _(stars,line,X))
```

produces the following output from *Whatis*:
which shows that procedure `printfancy` has three arguments: the first option (`stars`), the second option (`time`), and the first (and only) define argument (`X`).

8 Creating new DELIGHT versions

This section considers the entire process of creating new application-specific DELIGHT versions. These are executable programs such as `DELIGHT.SPICE`, `DELIGHT.MIMO`, etc., which contain all of the basic DELIGHT software plus other routines to, say, interface to a simulator or other scientific software. For example, a version interfaced to a simulator could allow results of simulations to be used in cost and constraint procedures for DELIGHT optimization. Similarly, a version interfaced to a matrix manipulation package could extend DELIGHT so that new matrix computations could be performed interactively or inside Rattle procedures. Throughout the subsections of section 8, we refer to your DELIGHT version as `DELIGHT.VNAME; VNAME`, standing for "version name", will actually be the name you choose for your version.

The nature of the material in this section and some of its machine-dependencies dictate that this section break from the style of previous sections by not containing, per se, interactive commands and responses for you to try out. For instance, commands to link/load executable programs are usually very different on different operating systems. However, most of the material in this section does not depend on your operating system; when it does, mention is made of that fact.

Sections 8.1 and 8.2 explain, respectively, the two most fundamental requirements for creating new DELIGHT versions: how to add built-in routines that are callable from Rattle, and how to declare variables for Rattle access. Section 8.3 discusses several routines that are called internal to DELIGHT and that must be tailored to each particular DELIGHT version. How to load/link your DELIGHT version, how to make the required `memfile` (see below), and the different ways of starting the program are the subjects of sections 8.4, 8.5, and 8.6, respectively. To aid in debugging built-in routines added according to section 8.1, section 8.7 gives some hints that can help pinpoint where the trouble may lie. Finally, section 8.8 presents some general guidelines, successfully used during the development of existing DELIGHT versions, for putting together your new DELIGHT version.

Memfiles. A brief discussion of memfiles is necessary before proceeding with the following sections. A `memfile` is a rather large binary file which contains the values of every DELIGHT internal variable that need be restored in value in order to bring DELIGHT back to the exact state it was in when a `store` command was issued to create the memfile. Memfiles are read back by the `restore` command or when DELIGHT is started normally. In other words if you set some variables and creates some Rattle procedures, store into a memfile, quit DELIGHT, and restart it at a later time from your memfile, then all of your variables and Rattle procedures will exist just as they did before you stored into the memfile.

This ability to restore the state of DELIGHT to what it was when a memfile was stored

---

1 On some computer systems, what we call binary files are sometimes called direct access, random access, or non-ASCII files. They cannot be printed out or edited and are only accessed through internal DELIGHT machine-dependent primitive routines.
DEUGHT For Intermediates (3/1/85) 55

has profound application. Since most DEUGHT commands are define/Rattle-procedure
pairs as illustrated in section 3 (recall, e.g., \texttt{define (showalgo,showalgo())}), the ability
to use such commands requires that DEUGHT read their defines and compile their pro-
cedures. Without the ability to restore DEUGHT's state, every command that a user
wanted to use would have to be processed in this way every time DEUGHT was started,
a very time-consuming task to ponder\(^2\). But with the \texttt{store} command, all of the standard
commands, the matrix macros, etc., can be processed just once and a standard
public memfile created by system personnel, a process referred to as "making a new
memfile". Then, by having DEUGHT start from this memfile, everyone has access to all
the commands, macros, defines, procedures, and variables that existed just before the
memfile was stored, i.e., when the memfile was "made". Section 8.5 discusses further
the process of making a version-specific memfile that other users of your version of
DEUGHT can access.

8.1 Adding Built-in Routines

This section describes how to add existing Fortran\(^3\) routines to DEUGHT so that they
are callable from Rattle procedures with exactly the same syntax as Rattle procedures
themselves are called. These routines might be simulation interface routines for a par-
ticular simulation program, utility routines, library routines, or routines containing
any computation whatsoever which needs the greater run-time efficiency of Fortran
over Rattle. Note that another option is to translate the Fortran routines into Rattle.
This translation could be computationally more costly since programs written in Rattle
usually run slower than their Fortran equivalents. In addition, translating a subroutine
into Rattle could be costly in terms of programmer time since Fortran routines are
often structureless and may be next to impossible to translate into the structured,
"goto-less" Rattle language. Thus, such translation should be avoided.

The addition of a new built-in routine to DEUGHT requires three operations:

1. make DELIGHT aware of the routine,
2. allow DELIGHT to call the routine, and
3. load/link the routine with DELIGHT.

To make DELIGHT aware of a new routine, a one line entry is added to file \texttt{anames},
which should reside in the directory where your version-specific memfile is to be made
(see section 8.5—\textit{Making a Memfile}). This entry associates a Rattle name with the rou-
tine and consists of the name by which the new routine is going to be known to Rattle
and the number of arguments to the routine. The Rattle name need not be the same as
the actual Fortran name. However in general, a good idea is to use either the Fortran
name (perhaps ending in an underscore thus making it a "system entity" to avoid name
clashes with names that might be used by users of your DELIGHT version) or a more
explanatory name. To allow DELIGHT to call a new routine, a call to the new routine is
added to Ratfor subroutine \texttt{abuilt} (for "application built-in"). All the calls in \texttt{abuilt}

\(^2\) Historically, it was Tommy Essebo's furious assertion to Bill Nye one summer—that it took over 15
minutes to start DELIGHT (just for the reason above)—that originally led to the creation of the \texttt{store}
and \texttt{restore} commands.

\(^3\) In this discussion, we often use "Fortran" to indicate any language whose normal programming cycle
consists of compile, link, and execute phases. In the context of DELIGHT, the language would probably be Rat-
for [2] or possibly C [3] although we, in particular, usually avoid using the name "Ratfor" due to the possible
must be in one-to-one correspondence with the entries in file *anames*. Finally, the procedure for load/linking a new routine with DELIGHT is highly system-dependent and will be covered in section 5.4.

As an example of the first two operations, suppose we wish to build into DELIGHT the two Fortran subroutines *clrnum* and *clrden*, each having no arguments. The names by which these are known to Rattle can be arbitrary but in our case, we let them be known by the self-explanatory names *ClearNumerator* and *ClearDenominator*. Thus, in file *anames* we would have

```plaintext
  ClearNumerator 0
  ClearDenominator 0
```

while Ratfor subroutine *abuilt* would simply require a computed goto entry (based on argument *funcno*, the entry number) and a call statement for each. A "conceptual" version of this subroutine would appear:

```fortran
  subroutine abuilt (funcno)
    go to (1,2), funcno
  1 call clrnum
     return
  2 call clrden
     return
  end
```

After this subroutine had been compiled and load/linked with DELIGHT, a memfile created, and DELIGHT started from this memfile, a user could type *ClearNumeratorQ* to have subroutine *clrnum* execute and *ClearDenominatorQ* to have *clrden* execute.

In reality, subroutine *abuilt* would be a bit more complex than this. Other things it must handle include passing arguments to the built-in routines, returning a function value from a built-in routine that is to act like a function in Rattle expressions, and special considerations for passing and receiving back integer arguments. Integer arguments are a consideration because all variables in Rattle are presently double-precision floating-point numbers. Thus to pass integer arguments, the Rattle double-precision arguments must either be copied into temporary integers, copied back to double from temporary integers, or both. For these purposes, there is a large work array called *iwork* (see below) that can be used for this temporary copying. Subroutine *rcopyi (D, I, N)* can be used to copy *N* items from double-precision array *D* to integer array *I*. Similarly, subroutine *icopyr (I, D, N)* can be used to copy integers back into double-precision arrays. When assigning to scalar integer temporaries from double-precision arguments, DELIGHT function *iround* should be used to round the doubles and avoid roundoff errors. These techniques are shown in the example below.

Inside subroutine *abuilt*, Rattle arguments are received via the Fortran double-precision array *rarray*, with the first argument in *rarray(e1)*, the second in *rarray(e2)*, etc. For double-precision arguments of a built-in routine, *rarray* can be used to simply "pass the Rattle arguments through", as shown in the example below. For integer arguments, as mentioned in the previous paragraph, *rarray* entries must be copied into or out of integer temporaries. To have a built-in routine return a function value, *rarray(retpt)* is assigned the value to be returned.

confusion between "Ratfor" and "Rattle".
To give a brief example of the other features of subroutine \texttt{abuilt} and the above argument techniques, we now consider another example with two built-in routines. The first is to be known as \texttt{FuncExamp} to Rattle, have Fortran name \texttt{funcex}, and return a double-precision function value with one double-precision argument. The second is to be known as \texttt{ProcExamp} to Rattle, have Fortran name \texttt{procex}, and have the twelve arguments shown below. These arguments consider all the various combinations of argument types: input only (read from but never written onto), output only (only written onto), and input/output (both read from and written onto), as well as scalars and arrays, both integer and double-precision:

\begin{verbatim}
1 - double-precision scalar input
2 - double-precision scalar input/output
3 - double-precision scalar output
4 - double-precision array input \hspace{1em} (size N_s)
5 - double-precision array input/output \hspace{1em} (size N_s)
6 - double-precision array output \hspace{1em} (size N_s)
7 - integer scalar input
8 - integer scalar input/output
9 - integer scalar output
10 - integer array input \hspace{1em} (size N_{10})
11 - integer array input/output \hspace{1em} (size N_{11})
12 - integer array output \hspace{1em} (size N_{12})
\end{verbatim}

For this example, file \texttt{anames} would contain

\begin{verbatim}
FuncExamp 1
ProcExamp 12
\end{verbatim}

while, with "\ldots" indicating other Ratfor code not shown here for clarity, subroutine \texttt{abuilt} would contain

\begin{verbatim}
subroutine abuilt (funcno, ..., retp, ..., iwork, ...)  
go to (1,2), funcno  
1 rarray(retp) = funcex ( rarray(e1) )  
   return  
2  i7 = iround (rarray(e7)) \# Copy inputs.  
i8 = iround (rarray(e8)) \# (i7, i8, i9, and iwork are temporaries.)  
call rcopyi (rarray(e10), iwork(1), N_{10})  
call rcopyi (rarray(e11), iwork(1+N_{10}), N_{11})  
call procex ( rarray(e1), rarray(e2), rarray(e3), rarray(e4), rarray(e5), rarray(e6), i7, i8, i9, iwork(1), iwork(1+N_{10}), iwork(1+N_{10}+N_{11}) ) \# 10 11 12
\end{verbatim}
Because the arguments to built-in subroutines and functions can only be double-precision or integer, modifications to the built-in routines themselves may have to be made. In Fortran, any arguments that are of type real must be converted to double-precision, to conform to the double-precision arguments which are passed from `abuilt`. This is easily done in some cases by putting an `implicit double precision (a-h.o-z)` statement at the beginning of each built-in Fortran routine, which will change the implicit typing for all real variables to double-precision. Any explicit real declarations such as `real v(10)` (as opposed to `dimension v(10)`) must be changed to double-precision as `double precision v(10)`.

Calls to subroutines/procedures in languages such as C can also be added to subroutine `abuilt`. You should pay attention to any machine-dependent procedure naming conventions that exist on your computer. For example, under UNIX, a Fortran routine that calls, say, C procedure `abc` must be named `aoc_` in the C source code.

Before closing we mention that everything in this section also applies to the subroutine/file pair `ubuilt/unames`, allowing ordinary users of any DELIGHT version to add their own built-in routines.

8.2 Declaring Variables for Rattle Access

When using existing routines which have been incorporated into DELIGHT, it may be necessary to access some of their variables. For example, many Fortran programs use common blocks as a means of passing or receiving information. To avoid having to make extensive modifications to these routines when they are built into DELIGHT (e.g., in order to set or get the value of these common block variables through subroutine arguments), you need to be able to directly access the variables in Rattle statements. This can be done by creating a special Fortran subroutine which contains calls to DELIGHT `variable-declaration` routines that associate each Fortran variable with a Rattle variable name. For example, variable `pdebug` discussed extensively in section 7.2.3, is declared in this way; when `pdebug=1` is typed, a Fortran common block variable is set which is tested by the Rattle parser to determine if debug printout is desired. There can be any number of Fortran subroutines containing calls to the variable-declaration routines. However, see the discussion of `dvdecs` in the next section: basically, all calls to declare variables should be executed when `dvdecs` is called internally.

The Rattle and Fortran variable names need not be the same. However, as for built-in routine names in the previous section, it is a good idea to use either the Fortran name (perhaps ending in an underscore, to make it a "system entity") or a more explanatory name. Another idea is that the Rattle names end in "_F" or "_FL", for example, to act as a reminder that they are Fortran declared variables.

The declaration subroutines are described in the following table. For each case, the name in quotes, which must end in a dollar sign ("$") string terminator, is the Rattle variable name. Scalar variables and arrays declared with these routines become
members of the pool of nonlocal Rattle variables.

<table>
<thead>
<tr>
<th>Subroutine Call</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>call deci ('NAME$', ,ivar)</td>
<td>Declares Fortran integer variable ivar.</td>
</tr>
<tr>
<td>call decia1 ('NAME$', ,iary,N1)</td>
<td>Declares Fortran integer array iary, having the one dimension N1.</td>
</tr>
<tr>
<td>call decia2 ('NAME$', ,iary,N1,N2)</td>
<td>Declares Fortran integer array iary, having the two dimensions N1 and N2.</td>
</tr>
<tr>
<td>call decia3 ('NAME$', ,iary,N1,N2,N3)</td>
<td>Declares Fortran integer array iary, having the three dimensions N1, N2, and N3.</td>
</tr>
<tr>
<td>call decr ('NAME$', ,rvar)</td>
<td>Declares Fortran real (double-precision) variable rvar.</td>
</tr>
<tr>
<td>call decra1 ('NAME$', ,rary,N1)</td>
<td>Declares Fortran real (double-precision) array rary, having the one dimension N1.</td>
</tr>
<tr>
<td>call decra2 ('NAME$', ,rary,N1,N2)</td>
<td>Declares Fortran real (double-precision) array rary, having the two dimensions N1 and N2.</td>
</tr>
<tr>
<td>call decra3 ('NAME$', ,rary,N1,N2,N3)</td>
<td>Declares Fortran real (double-precision) array rary, having the three dimensions N1, N2, and N3.</td>
</tr>
</tbody>
</table>

In the array declarations above, the dimensions should be identical to those of the actual Fortran array.

The following example of the special built-in Fortran subroutine needed to make calls to the above declaration routines contains examples of those routines:

```fortran
subroutine Vinit
  double precision xvar, xarray
  common /cname/ ivar, iarray(200), xvar, xarray(10,20)
  call deci ('iver-P-i', ivar)
  call decia1 ('iarray-F-S*, iarray, 200)
  call deer ('xvar-F-J', xvar)
  call decra2 ('xarray_F_l', xarray, 10, 20)
return
end
```

Since declared Fortran variables exist in the pool of nonlocal variables, they are accessed in Rattle procedures by importing them. For example, the following Rattle procedure uses the variables declared above:
procedure SetFortranVars
import ivar_F_, iarray_F_, xvar_F_, xarray_F_
ivar_F_ = ...
for k = 1 to 200
  iarray_F_(k) = ...
}

If it is desired to make a declared variable global so that it does not need to be imported, the Fortran subroutine such as Vinit above can have a call decglo ('NAMES') statement after the normal declaration call. In the above example, to make Rattle variable xvar_F_ global, we would have:

call deci ('ivar_F_'), ivar
call decglo ('ivar_F_')

An important restriction on how declared Fortran variables can be used in Rattle is that they cannot be passed as arguments to Rattle procedures. They should instead be imported or made global, as shown above.

8.3 Version-Specific Routines Called by DELIGHT

There are several routines that get called by DELIGHT automatically when various actions or operations occur. For example, when any DELIGHT version starts, subroutine dvinit is called to allow any version-specific initialization to occur. All you (the creator of the DELIGHT version) do is put into subroutine dvinit anything that must get executed once when DELIGHT first starts up. This might include, for example, certain variable initializations that might, say, read from a file, or the one-time setup of a runtime dynamic memory manager. If there is no initialization of this sort, then subroutine dvinit need not be defined; a dummy subroutine containing just a Fortran return and end is used by default. This is true for all of the routines discussed in the remainder of this section.

The version-specific routines and some version-specific files are summarized in the following tables:
### DEUGHT Version-Specific Routines

<table>
<thead>
<tr>
<th>Name</th>
<th>When called</th>
<th>What it should do</th>
</tr>
</thead>
</table>
| dvdecs | Program startup | All calls to declare variables for Rattle access (see section 8.2) should be executed. Thus, as mentioned in section 8.2, if there are several Fortran subroutines which contain calls to the variable-declaration routines, each of them should be called inside your subroutine `dvdecs`:

```fortran
subroutine dvdecs
  call Fdecs1 ('n-F_S', n)
  call Fdecs2 ('z-F_S', z, n)
  ...
return
end
```

<table>
<thead>
<tr>
<th>dvexit</th>
<th>Program termination</th>
<th>This is called right before DEUGHT finishes executing, e.g. after a <code>quit</code> command is typed, and should contain whatever cleanup or final screen messages are necessary for the version.</th>
</tr>
</thead>
<tbody>
<tr>
<td>dvinit</td>
<td>Program startup</td>
<td>Whatever version-specific initialization is required should be performed. There is an argument that is <code>YES</code> (defined as 1 in file <code>style</code>) if the startup is <code>forced</code> and you are making a new memfile. It is <code>NO</code> (defined as 0) if you are simply starting DEUGHT normally (unforced) from an existing memfile. By testing this argument, you can have certain initializations occur only for either type of DEUGHT startup. See section 8.5 for more on forced versus normal startups.</td>
</tr>
<tr>
<td>dvname</td>
<td>Program startup</td>
<td>The name of this DEUGHT version should be returned, i.e., the name to be appended to <code>mem</code> for default memfiles, to <code>img</code> for login messages, etc. For example, if this routine returns <code>MIMO</code>, then this version is <code>DELIGHT.MIMO</code> which starts up by reading memfile <code>&lt;mem.MIMO&gt;</code> (see section 8.5) and prints the “Welcome to DELIGHT.MIMO” login message contained in file <code>&lt;img.MIMO&gt;</code>, etc.</td>
</tr>
<tr>
<td>memfio</td>
<td>During a <code>store</code> or <code>restore</code> command</td>
<td>The name <code>memfio</code> is an acronym for “memfile-io” since the purpose of this routine is to allow version-specific internal (Fortran) variables to be written out to and read in from a memfile. By using calls to the routines shown in the next table, these variables' values are stored in a memfile and can thus be restored (when DEIGHT is started from the memfile) to their exact state before the <code>store</code> command was issued. This was explained in the introduction to section 8.</td>
</tr>
</tbody>
</table>
### DELIGHT Version-Specific Files

<table>
<thead>
<tr>
<th>Name</th>
<th>When read</th>
<th>What it contains</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;lxngVNAME&gt;</code></td>
<td>After a restore command</td>
<td>This file, automatically printed on the terminal screen after a restore command, usually contains a &quot;Welcome to DELIGHT.VNAME&quot; message. It is your responsibility (as system personnel) to insure that the system openhdtl file (see section 5.2.1) is set up so that users can access this file.</td>
</tr>
<tr>
<td><code>&lt;memVNAME&gt;</code></td>
<td>Program startup</td>
<td>This is the default memfile that is read when version DELIGHT.VNAME is started (see section 8.6). As above, the system openhdtl file must be set up so that users can access this file.</td>
</tr>
<tr>
<td><code>&lt;HsVNAME&gt;</code></td>
<td>During a help command</td>
<td>When one of the help commands or section 2 is executed inside DELIGHT.VNAME, this binary helper file is opened to look for the command or topic requested. See The Helper Facility [8].</td>
</tr>
</tbody>
</table>

---

**More on Subroutine Dvname.** Subroutine `dvname` is probably the simplest routine ever written in any programming language! All you have to do is return the version name in the (one) argument as done in the sample routine below:

```fortran
subroutine dvname (str)
   integer str
   call pcopys ('SPICES', str)
   return
end
```

Of course, the above `dvname` would be for version `DELIGHT.SPICE`. Be sure to include the dollar sign ("\$") quoted-string terminator on the `pcopys` call.

**More on Subroutine Memfio.** Both output to and input from a memfile are performed by subroutine `memfio`. It contains the single argument `mode` to indicate whether the memfile is being written or read:

```fortran
mode = 1    -- Read in memfile variables.
mode = 2    -- Write out memfile variables.
```

The basic format of subroutine `memfio` is:

```fortran
subroutine memfio (mode)
   integer mode
   common /BLOCKNAME1/ ivar1, ivar2, ...
   common /BLOCKNAME2/ ...
   ...  (Common blocks containing variables to be written to and read from the memfile.)
```
if ( mode .eq. 2 ) go to 2
  call rbini (ivar1)
  call rbini (ivar2)
  ...
  go to 9
2  call wbini (ivar1)
  call wbini (ivar2)
9  return
end

(Read in the variables, in the same order as below.)

(Write out the variables, in the same order as above.)

Routines *rbini* and *wbini* are for reading and writing a single integer to/from the memfile, respectively. Real variables and arrays of both types can be handled with the routines in the following table:

<table>
<thead>
<tr>
<th>Memfio Subroutines For Reading and Writing To/From a Memfile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutine Call</td>
</tr>
<tr>
<td>call rbini (ivar)</td>
</tr>
<tr>
<td>call rbinia (iary, size)</td>
</tr>
<tr>
<td>call rbini (rvar)</td>
</tr>
<tr>
<td>call rbinra (rary, size)</td>
</tr>
<tr>
<td>call wbini (ivar)</td>
</tr>
<tr>
<td>call wbini (iary, size)</td>
</tr>
<tr>
<td>call wbini (rvar)</td>
</tr>
<tr>
<td>call wbini (rary, size)</td>
</tr>
</tbody>
</table>

Note that no separate routines are provided for arrays with more than one dimension; the product of the dimensions can be passed as the size argument. For example, if you had real array *rx*(3,5) then you could use the calls *call rbinra (rx,5*5)* and *call wbina (rx,5*5).*
8.4 Loading DELIGHT

The commands required to load/link your DELIGHT version are highly machine-dependent. However generally, after you have compiled: (1) any of the version-specific routines detailed in section 8.3, (2) any routines to be considered built-in per section 8.1, and (3) any other routines that may be called by the ones just mentioned such as the internal routines of a simulation program, you would then give a command similar to the following:

```
LOAD name=DELIGHT.VNAME OBJECT1 OBJECT2 ... DLIB1 DLIB2 ...
```

where "DEUGHT.VNAME" is the name of your DELIGHT version's executable program file, "OBJECT1", "OBJECT2", etc. are the names of all the object files produces by the compilations mentioned above, and "DLIB1", "DLIB2", etc. are the names of DELIGHT object file libraries that contain all of the "core" (non-version-specific) DELIGHT routines. After the executable program has been successfully loaded, the next step is to make an associated memfile, the subject of the next section.

8.5 Making a Memfile

Memfiles were briefly introduced at the end of section 8. It was pointed out that they are created (and written) by the store command but read back by either the restore command or when DELIGHT is started normally. To clarify, when DELIGHT is started, a restore is (internally) performed automatically in order to restore all DELIGHT internal variables and thus start the user off in the state in which all the standard commands, macros, etc., exist and are immediately ready for his use. This restore is usually from a public memfile that is not in the user's directory. However, as shown in section 8.6, a memfile can be specified as an argument on the command line used to start DELIGHT.

If a user works in DELIGHT for a time, creating several commands, Rattle procedures, variables, and so on, he may wish to store into his own memfile so that he can restore his current state on, perhaps, the following day. The format of the store command is

```
store [ MEMFILENAME ] [ 'IDENTIFIER' ]
```

where the optional MEMFILENAME argument is the name of the memfile (if unspecified, the name "memfile" itself is used) and IDENTIFIER is an optional identifier quoted string which can serve to identify some characteristics of the current state being stored into the memfile. For example, the following are all valid store commands:

```
store (This uses filename "memfile".)
store memtemp
store memdebug 'Partially debugged procedure matcal()'
store memBASIC 'Standard Optimization Memfile with Matrix Macros'
store basicfile
```

The last command above would produce the following warning message:
WARNING: Memfile filenames should start with "mem" by convention.

This convention exists because memfiles can be very large and you probably don't want to have large files "sitting around" when they are not needed; when you examine a (hopefully alphabetized) listing of all your filenames, we want you to be able to spot memfiles easily so that you can remove any that are not needed.

The IDENTIFIER string given on the store command gets printed to the screen whenever a restore is performed. Also, there is a command called memdate that shows the actual MEMFILENAME argument given on the store command, and the date the command was given, for the memfile last restored from. Thus, your terminal screen might appear:

```
% DELIGHT.VNAME
DELIGHT: Restoring from <memVNAHE> ...
Identifier: Standard VNAME Memfile with Optimization

***** Welcome to DELIGHT.VNAME *****


Developed by the Optimization-Based Computer-Aided-Design Group
University of California
Berkeley, Ca. 94720.

Copyright 1983 by the Regents of the University of California. All Rights Reserved.

1> history
2 store ../memfiles/memnew 'Standard VNAME Memfile with Optimization'
1> mandate
Memfile "../memfiles/memnew" stored on 01/28/85 at 06:35:57
1> restore memdebug
Restoring from memdebug ...
Identifier: Partially debugged procedure matcal()

***** Welcome to DELIGHT.VNAME *****

```

The above shows an identifier and the result of a memdate command for each of two different memfiles. You should also notice that the history command shows the store command that was used to create the memfile. As shown in the first usage, this is true even if you did not issue the store command yourself.

We now are ready to illustrate how system personnel make a new memfile. As stated above, when DELIGHT is started "normally", a restore from a public memfile is performed automatically. Also, we have shown how a user can create his own memfile with the store command (after DELIGHT has been started normally). But how is the public memfile created initially, when as yet no memfile exists? Equivalently, how can system
personnel start DEUGHT without the public memfile so that the store command can be given to create the memfile? This is the purpose of the -force option to DEUGHT: If DEUGHT is started with the (operating system) command

\[ \text{DEUGHT} -\text{force} \]

then it will start up without any memfile whatsoever and begin in a state in which the barest minimum of commands, macros, etc., exist\(^4\). Before DEUGHT presents the "1>" prompt to the terminal screen, however, it checks if a file exists called setup and if it does, it is also automatically included.

**The Setup File.** Starting DEUGHT with the -force option and having an appropriate setup file, then, is how a memfile can be created from scratch. Suppose you want to create a memfile called memplot which contains only the plot command (as well as the minimum commands mentioned above). You would use the following setup file

```plaintext
include <plot>
store memplot
quit
```

and start DEUGHT with the -force option as seen below:

```
% DELIGHT.VNAME -force
DELIGHT.VNAME: Beginning forced startup ...
Almost ready ...

***** Welcome to DELIGHT.VNAME *****

Storing into memplot ...
Goodbye Whoever_You_Are, It is 18:12:11, Date 01/29/85.
```

After the "Welcome" message, DEUGHT automatically includes the setup file, which includes system file <plot>, performs the store into memfile memplot, and exits DEUGHT with the quit statement. If you (or another user of your DEUGHT version) now start DEUGHT with this memfile by typing

```
DELIGHT.VNAME memplot
```

then the plot command (as well as the minimum commands mentioned above) is available to you immediately.

Of course, the memfile that most users will start from should contain many more commands than just plot. Either you (as the one making the public memfile) can include all desired commands individually as in the example setup file:

\[ \text{include <standdfs>} \text{ and <stanstu>} \text{ are available in this minimum state. In fact, DEUGHT, when started with the -force option, simply pushes back the statement include <standdfs> (which itself includes file <stanstu>).} \]

\(^4\) To be precise, all of the defines and procedures in system files <standdfs> and <stanstu> are available in this minimum state. In fact, DEUGHT, when started with the -force option, simply pushes back the statement include <standdfs> (which itself includes file <stanstu>).
or you can include one of a few standard system "setup" files that themselves include all the standard DELIGHT commands, macros, etc. Including file `<macdefs>` brings in all the matrix macros such as `matop`, `det`, `clip`, `fill`, `timeq`, `quadprog`, etc. Including file `<Esetup>` brings in the matrix macros, all other standard DELIGHT commands, and all optimization-related commands such as `solve`, `run`, `initprob`, `testgrad`, etc. Hence the simplest setup file for creating the public memfile would be:

```
include <Esetup>
store <memNAME> 'Standard VNAME Memfile with Optimization'
quit
```

There are several other things that usually go into real setup files. These include:

- Includes for other files besides `<Esetup>`. These might be necessary, for example, to implement version-specific commands such as for running a simulator for a particular type of engineering design.

- A `terminal` command to set the default graphics terminal type to the most commonly used terminal.

- Various `store` commands throughout the setup file that create incomplete memfiles containing all that has been Rattle compiled up to that point. These save you from having to remake the whole memfile in the event that DELIGHT aborts; you simply restart DELIGHT from the latest successfully stored memfile and include a scratch file in which you place the unprocessed portion of the setup file. A good idea is to place identifiers on these `store` commands as in

```
store memtemp 'Finished <Simcmds> (INCOMPLETE MEMFILE)'
```

- `set_option` commands to set various DELIGHT options. Two of particular importance are set by the following commands:

```
set_option DOptions -LineNumTrace = YES
set_option DOptions -SaveLocals = YES
```

Throughout DELIGHT For Beginners [4] and this document, whenever an interrupt in Rattle execution is shown to occur, you can always type `trace` and see what procedures have been called on what file line numbers. Since these line numbers for every procedure take a bit of storage in a memfile, the default is to `not` store line numbers; the first `set_option` above turns on their storage. Similarly, local variables, arrays, etc., of procedures are by default, not stored in the DELIGHT internal symbol table and hence into a memfile; the second `set_option`

---

5 There is no "setup" file for including just the matrix macros and all standard commands since, after all, DELIGHT is for optimization.
above turns on their storage. You would want to place these statements after an
\textit{include <Esetup>} but before any includes of files containing Rattle procedures
that need to be debugged.

- A \texttt{solve} command to presolve a particular optimization algorithm that will
definitely be used for all optimization in your DELIGHT version. An example of
this is shown below.

- A \texttt{clear_time} command, discussed in \textit{DELIGHT For Beginners}, to reset all the
call-counts and the cpu time values to zero that are displayed by the \texttt{display_time}
command. This will avoid having the latter command display to an ordinary user
cpu times associated with things that occurred when the public memfile was
created; \texttt{display_time} should show the user's own cpu time only.

- A \"%Z\" to close the helper binary file. This is explained in [6]. Suffice it to say
here that if you are not generating your own help entries, the \"%Z\" will not cause
any problem if it is there.

There are two other matters that concern including files. First, instead of just
unconditionally including a file, it is a good idea to test whether some entity (pro-
cedure, define, macro, etc.) created inside the file exists and include the file if it does
not. This is done using \texttt{if\_NOTHERE}, which has the syntax shown by the following
example:

\begin{verbatim}
if\_NOTHERE output\_to then include <output>
\end{verbatim}

By using \texttt{if\_NOTHERE}, you avoid having the same file included twice, since the first
inclusion would create the entity tested for. Second, files can be included with
\texttt{include\_and\_print} instead of with \texttt{include}. This causes the filename and the total
DELIGHT program execution time to be printed when the inclusion of the file first
begins. To determine how much cpu time was consumed during an \texttt{include}, this cpu
time would be subtracted from the next time. The \texttt{include\_and\_print} statement also
indents the filename if this file is included by the previous. For example, suppose we
have the following three files:

\begin{verbatim}
File t: include\_and\_print t1
       include\_and\_print t2

File t1: include\_and\_print t11
       include\_and\_print t12

File t2: (empty)
\end{verbatim}

Then if we typed \texttt{include\_and\_print t}, we would see (except for different cpu times):

\begin{verbatim}
1> include\_and\_print t
including t             (206sec)
including t1            (209sec)
including t11           (211sec)
including t12           (212sec)
including t2            (215sec)
\end{verbatim}
From this output, you can immediately tell that file $t_1$ includes files $t_{11}$ and $t_{12}$ and that file $t$ includes files $t_1$ and $t_2$. This knowledge can be important in tracing down compiler error messages.

We now present a complete setup file that includes several of the ideas discussed above.

#### DELIGHT.SPICE memfile setup ####

store memfile 'The very beginning (INCOMPLETE MEMFILE)'
include_and_print <Setup> ## Standard DELIGHT optimization setup.
terminal hp ## Set default terminal.
set-option DLoptions "nnkevhelp=YES ## If "-nnkevhelp" option given,
        ## turn on making binary help file.
if NOTHERE dslv then include_and_print <ddisplay>
if NOTHERE interpolated_array then include_and_print <itparray>
store memtemp 'Everything up to "use <3imall>" (INCOMPLETE MEMFILE)'
include_and_print <3imall> ## DELIGHT.SPICE Simulation Interface,
include_and_print <Simckt> ## setckt command.
include_and_print <Simtryv> ## tryv command.
store memtemp 'Before algorithm (INCOMPLETE MEMFILE)'
set_option DLoptions "LineNumTrace = YES
set_option DLoptions "SaveLocals = YES
solve using Addmld ## Feasible-direction-multicost algorithm
        ## with lumped finite differences.
clear_time ## Clear times for "display_time".
S## Close help file.
store <memSPICE> 'Spice Basic Memfile with Precompiled Phase I-II-III Algorithm'
quit

One final reminder, as pointed out in section 8.1, to make DELIGHT aware of new routines being made built-in, file $anames$ is used. Since this file is read by DELIGHT during a forced startup, it should reside in the same directory where you run DELIGHT to make your memfile.

### 8.6 Starting DELIGHT

As covered in the previous section, when DELIGHT is started with the -force option, it automatically includes a file called setup. Similarly, when DELIGHT is started normally (i.e., without the -force option), it automatically includes file startup if it exists just after the "Welcome to DELIGHT" message. This allows you to have commands execute automatically which you would otherwise have to type when first starting DELIGHT. One common entry in the startup file is a line such as $user_name_is Pokey$ which tells DELIGHT your name.

Another property of how DELIGHT is started has been alluded to in earlier subsections of section 8. This concerns which memfile is used during a normal startup. The
rule is simple. First, if you specify a memfile as in

```
DEIGHT memfilename
```

then DEIGHT will read from the specified memfile. If you don't specify any memfile, then DEIGHT will read from file `memfile` if it exists; if it does not exist, then DEIGHT reads from file `<memVNAME>` where `VNAME` is substituted by the version name declared in subroutine `dvname` (see section 8.3), e.g., `SPICE` for DEIGHT.SPICE. For this purpose, the basic DEIGHT version has version name `BASIC`. Thus, typing `DEIGHT` alone, if there is no file `memfile`, will restore from memfile `<mem.BASIC>` (and print the login message from file `<lmg.BASIC>`).

All of the preceding is summarized in the following table:

<table>
<thead>
<tr>
<th>Starting DEIGHT</th>
<th>Command Option</th>
<th>File Automatically Included</th>
<th>Memfile Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forced startup</td>
<td><code>DEIGHT.VNAME -force</code></td>
<td>setup</td>
<td>(none)</td>
</tr>
<tr>
<td>Normal startup</td>
<td><code>DEIGHT.VNAME MemFileName</code></td>
<td>startup</td>
<td>MemFileName Memfile (or) <code>&lt;memVNAME&gt;</code></td>
</tr>
</tbody>
</table>

There are several other options that can be used when starting DEIGHT. These are explained in the options sections of the help entry for DEIGHT (which may be obtained by typing `helpoptions DELIGHT` while in DEIGHT) and are summarized below:

- **-echo** Immediately turn on the echoing of all lines read by DEIGHT.
- **-force** Start DEIGHT without any memfile, and automatically include file `setup` (unless changed by the `-l` option below).
- **-fix** If the message "Bad file memfile" is seen, restarting DEIGHT with this option will attempt to fix the bad addresses stored in the memfile (see below).
- **-verbose** Print out chatter showing what is going on during either a forced or normal startup.
- **-makehelp** Cause "%N", "%U", help lines in file `setup` (or a file included by it) to generate the binary help file; without this option, these lines are simply ignored.
-makevhelp  Set an internal DELIGHT flag so that turning on help via
   set_option DLOptions "makevhelp=YES"
say, in a setup file, causes succeeding "%N", "%U", help lines to gen-
erate a binary help file—just as if you had started DELIGHT with
the -makehelp option or did a
   set_option DLOptions "makehelp=YES"
Thus, if set_option DLOptions "makevhelp=YES" appears in a setup
file, it turns on the generation of a binary help file only if DELIGHT
was started with -makevhelp.
-trsorc  Print the source directory for all <FILENAME> files opened, pro-
ducing the same output as the whereis command presented in sec-
tion 7.3.
-DXXXX  Substitute filename XXXXX for setup during a forced startup or for
start up during a normal startup.

The -echo, -verbose, and -trsorc options are useful for debugging a forced DELIGHT
startup. The -echo option will cause every single line read by DELIGHT to be echoed to
the screen; try it if you want to see and understand why making a memfile takes so
long!

The -verbose option is much less "verbose" than -echo; it simply prints out what is
going on internally during either type of DELIGHT startup. Below is shown its output for
a forced startup:

```
DELIGHT.VNAME -force -verbose
DELIGHT.VNAME: Beginning forced startup ...
   -verbose: Reading files <exops> and <exfuns>.
   -verbose: Reading file <rrdata>.
   -verbose: Declaring Rattle-access variables.
   -verbose: Version-specific Rattle-access vars.
   -verbose: Including file <standefs>.
Almost ready ...
   -verbose: Including file <stanstuf>
Making public: stackf
Making public: ph_stackf
Making public: clear_stackf
   -verbose: Including file <helpmaca>
***** Welcome to DELIGHT.VNAME *****
...
```

1> quit
Goodbye Whoever_You_Are, It is 18:34:32, Date 02/03/85.

The fact DELIGHT comes back with its prompt above implies that there is not a setup
file in the present directory (or at least, the setup file does not contain its own quit
command).

Before the addition of the -fix option to DELIGHT, almost every time you relinked the
DELIGHT executable, you needed to make a new memfile. This is because a memfile
has Fortran addresses from an internal dynamic memory manager stored into it and
when you relink, these stored addresses of fixed DELIGHT internal variables probably
change. Hence, when you try to start DELIGHT from the bad memfile, you see some-
ting like the following:
The number 1027 is not important but shows the difference between the new address of a particular DELIGHT variable and its address as stored in the memfile. The word "more" above means that the new address is greater, i.e., that the DELIGHT executable grew by 1027 integer "words".

With the -fix option, however, DELIGHT attempts to fix (update) the addresses stored in the memfile as they are read. Thus, after seeing the "BAD file <memVNAME>" message, you can try to fix the memfile by using

```
DELIGHT -fix
```

If after this, DELIGHT either does not give a prompt or it does but aborts immediately or otherwise acts strangely, then the -fix option has failed and a new memfile must be made. This occurs, for example, if the basic DELIGHT common blocks change or if you add or remove variables from your subroutine memfio (see section 8.3).

The -ixxxxx option, for including another file other than file setup during a forced startup, allows you to have several different setup files. One could create a complete memfile while others, to decrease the time required to make the memfile, could create only partial memfiles, that is, without all the necessary files included. These partial memfiles would usually be for debugging (a built-in routine, for example). The -ixxxxx option is also useful during a normal startup to simulate "batch-like" operation of DELIGHT. For example, suppose file temp5 contained some sequence of commands whose execution could occur non-interactively such as:

```
echo_io_to temp5output
read_matrix ...
matop ...
printv ...
echo_io_end
quit
```

Then starting DELIGHT using

```
DELIGHT -ftemp5
```

would cause file temp5 to be included automatically. If your operating system allowed this command to be entered into a "batch" queue, then DELIGHT would execute the commands in file temp5 without user interaction.
8.7 Debugging Added Built-in Routines

After you've linked your new DELIGHT version together and created a memfile according to sections 8.1 and 8.5, you are ready to see if your new built-in routines work. For the example shown in section 8.1, they could be called directly as in either of the following:

```delight
1> FuncExamp(5)
1> print FuncExamp(7)
```

If your built-in routines did not print anything to the screen, the only way to tell if they were working would be to check the values of any arguments or the function value returned. As detailed in section 7.2.3, you can set variable `pdebug` to aid in the debugging process:

```delight
1> pdebug = 3
1> FuncExamp(7)
```

In a real debugging situation, you would examine carefully the values returned above. Function number 1001 tells you that this is the first function in `built`. (Similarly, 2001 would be for the first function in `ubuilt`, mentioned at the end of section 8.1.)

A very important problem when debugging built-in routines is what to do when one of them gets "hung," i.e., goes into an infinite loop and does not return to Rattle execution. In this case, pressing the special interrupt ("break") key twice to generate a hard interrupt (see [4]) will not suspend execution in the built-in routine. Basically, there is no way to suspend such infinite loops; DELIGHT must be aborted in some manner, additional print statements added inside the culprit routine, DELIGHT relinked, and possibly a new memfile created. However, to aid in tracing down the bug, the hard interrupt does execute a `trace` command so you can at least see what Rattle routines are involved and where the problem is in terms of Rattle execution. This is shown in the following trivial example in which built-in function `sdelay` has been used to simulate a hung built-in routine by delaying execution for 10 seconds:

```delight
1> function inner
1| sdelay(10)
1> function outer
1| inner()
1| outer() (Immediately press the "break" key twice)
```

WARNING: A second interrupt has been received before DELIGHT has detected the first ... DELIGHT is possibly hung in a built-in routine. A "trace" follows:
Interrupted IN procedure
inner (Input from the terminal)
Called by outer (Input from the terminal)
CONTINUING EXECUTION...

Interrupt...
1> reset
1>

If these procedures had been Rattle compiled by including them from a file, then the traces above would have shown the usual statement line numbers in the procedures.

8.8 General Guidelines for Creating a DELIGHT Version

This section contains a set of stylistic guidelines that have proven themselves well in creating new DELIGHT versions as well as in implementing the original DELIGHT system.

1. When creating Rattle procedures for various version-specific needs, make all related procedures have names that begin with a 2- or 3-letter (unused) acronym such as MTX_for matrix routines, GX_for extended graphics routines, etc. For example, you might have graphics procedures GX_addbox, GX_paintscreen, etc. These names make it highly unlikely that your new names will clash with any existing procedure names or with names used by a user of your DELIGHT version.

2. Avoid global variables when possible since they increase the possibility of name clashes with local variables of procedures created by users of your DELIGHT version.

3. For arrays and variables that are to be shared among several procedures in different files, set up a separate file containing a definition of this "data base", along with complete documentation in comments. This file is then included first in your setup file. For example, part of the data base associated with the Phase I-II-III Method of Feasible Directions optimization algorithm appears:

```
# DATA BASE

array X_name_(0,0), X_scale_(0), X_variation_(0), X_init_(0),
X_min_good_(0), X_min_bad_(0), X_min_type_(0),
X_max_good_(0), X_max_bad_(0), X_max_type_(0),
Most_good_(0), Most_bad_(0), Most_min_or_max_(0).

create Q_count_  # Used in <usrdefs2> and <simv>.
create MAXInitMesh_  # Maximum of the initially number meshes.
create Meshpts_updated_  # TRUE if the mesh points in arrays
# Fineq.meshpts(...) or Most.meshpts(...) have
# been updated (variable grid spacing);
# used and reset in <Egbmupdt>
```
array present_proc(PMAXTOKSIZEx) # Packed string: present procedure name.

create nchan_ # Output logical unit number for temporary file
    # shared by objective_(), for_every_(), and
    # constraint_().

define (HARD,1) # Possible values for Xmin-type_(), Ymax-type_(),
define (SOFT,2) # Ineq-type_(), and Fineq-type_().
define (MINIMIZE,1) # Possible values for Hcost_min_or_max_().
define (MAXIMIZE,2)
define (LINEAR,1) # Possible values for WFSpacing_(), WFlSpacing_() # for functional constraints.
define (LOGARITHMIC,2)

4. When expanding an array dynamically, use an increment greater than one for efficiency. For example, suppose you are reading in a system description from a user and you don't know a priori how many "blocks" he will enter. Then an inefficient piece of Rattle code might appear

```
blockcount = blockcount + 1
array BlockPtr(blockcount)
BlockPtr(blockcount) = ...
```

while it would be far more efficient to use

```
blockcount = blockcount + 1
if ( blockcount > arydim(BlockPtr) )
    array BlockPtr(blockcount+20)
BlockPtr(blockcount) = ...
```

Here, by expanding array BlockPtr by 20 instead of by 1, there are 20 times fewer expansions of the array then before (though there might possible be 19 wasted array elements at worst case).

5. Always zero the size of temporary local arrays at the bottom of a procedure, as shown in the following:

```
procedure examp {
    array work(100)
    ...
    array work(0)
}
```

This avoids having these arrays stored into a memfile if the memfile is stored after a procedure without the last array statement above executes.

6. Do not declare formal arrays that are never accessed with subscripts since it creates unnecessary run-time overhead. For example in the following, even though Need and NoNeed are both arrays when passed, the latter does not need to be declared since it is never subscripted in the body of the procedure:

```
procedure proc (Need, n1, NoNeed, n2) {
    array Need(n1)
    array NoNeed(n2)
    for i = 1 to n1
        otherproc (Need(i), NoNeed, n2)
}
```

7. Never remove (see DELIGHT For Beginners [4] for a discussion of the remove
command) a buggy procedure before recompiling it since this will destroy all calls to it in other procedures. This is because the other procedures will, of sorts, still be calling the removed procedure's Rattle intermediate code. For example, if you had the two procedures:

```plaintext
procedure proc1
...
procedure proc2
  proc1()
```

and you found bugs in procedure `proc1`, removing it with the `remove` command and then redeclaring it would leave procedure `proc2` still calling the removed version of `proc1`.

8. For portability reasons (assuming that some day you might want to port your DELIGHT version to another computer), always limit your filenames to a maximum of eight characters. Moreover, the names should not contain any special characters other than letters and digits and should begin with a letter. If your operating system requires, for example, all filenames to contain, say, a dot and a filetype extension, you can still use in your `setup` file the filenames without the dot and extension; a DELIGHT internal machine-dependent primitive should have been set up so that the required dot is added automatically. Thus, having

```plaintext
include_and_print clrdata
```

in your `setup` file might actually cause file `clrdata.ascii` to be included.

9. To give you an idea of a good directory structure for setting up your DELIGHT version, here is one that has been used for DELIGHT.SPICE:

```
doc/    Guide            (Documentation directory)
         (Beginners Guide)
make/    DELIGHT.VNAME   (Directory where DELIGHT is loaded and memfile made)
         Makefile       (Executable file)
         anames         (Built.r Rattle names)
         memfile        (Created memfile)
         openhdtl       (Openhdtl file)
         setup          (Setup file)
src/     (Directory containing all source)
         src/display/   (Subdirectory for display command)
         Makefile       (Subdirectory for display command)
         displa.o       (Compiler output object file)
         displa.r       (Target for source file for display
                         command)
         ...
src/include/     (Subdirectory for shared
                 "included" files.)
            cshare1     (First shared file)
            cshare2     (Second shared file)
            ...
```
Before setting this guide aside, you should be reminded of what temporary files have been created in your current directory (and thus can be removed) throughout the course of performing the boldface commands in this guide. They are files junk, junk1, junk2, junk3, junk12, junkxy, and myfile.

Acknowledgements

For their tremendous assistance in reviewing this user’s guide, we give special thanks to Mark Austin, Andrew Heunis, Elijah Polak, Alberto Sangiovanni Vincentelli, and Stephen Wu.

This work was supported by the National Science Foundation (NSF) under grant ECS-8121149, by the Air Force Office of Scientific Research (AFOSR) United States Air Force Contract No. AFOSR-83-0361, by the Office of Naval Research (ONR) under contract N00014-83-K-0802, by Microelectronics Innovation and Computer Research Opportunities (MICRO) under contract N00039-83-C-0107, and by a grant from the Semicon-
ductor Products Division of the Harris Corporation.

References


Index

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>@system</td>
<td>52</td>
</tr>
<tr>
<td>ABORT ABORT ABORT message</td>
<td>51</td>
</tr>
<tr>
<td>aborting on numeric overflow</td>
<td>50</td>
</tr>
<tr>
<td>abuild subroutine</td>
<td>58</td>
</tr>
<tr>
<td>accessing Fortran variables in Rattle</td>
<td>58</td>
</tr>
<tr>
<td>adjugate keyword to Matrix Func command</td>
<td>39</td>
</tr>
<tr>
<td>anames file</td>
<td>56, 69</td>
</tr>
<tr>
<td>&lt;anspromp&gt; file</td>
<td>24</td>
</tr>
<tr>
<td>answer_to_prompt procedure</td>
<td>24</td>
</tr>
<tr>
<td>answer_to_prompt modes</td>
<td>28</td>
</tr>
<tr>
<td>array out-of-bounds run-time error</td>
<td>44</td>
</tr>
<tr>
<td>Assignment Continuation Convention</td>
<td>42</td>
</tr>
<tr>
<td>Auto-Pushback Convention</td>
<td>15</td>
</tr>
<tr>
<td>backslash character</td>
<td>4, 28</td>
</tr>
<tr>
<td>BASIC DELIGHT version</td>
<td>8</td>
</tr>
<tr>
<td>batch mode operation of DELIGHT</td>
<td>73</td>
</tr>
<tr>
<td>binary help files</td>
<td>5, 8</td>
</tr>
<tr>
<td>box command</td>
<td>52</td>
</tr>
<tr>
<td>C language</td>
<td>58</td>
</tr>
<tr>
<td>clear_time command</td>
<td>88</td>
</tr>
<tr>
<td>close built-in routine</td>
<td>29, 48</td>
</tr>
<tr>
<td>Colon Convention in routine answer_to_prompt</td>
<td>26</td>
</tr>
<tr>
<td>commands:</td>
<td></td>
</tr>
<tr>
<td>box</td>
<td>52</td>
</tr>
<tr>
<td>clear_time</td>
<td>88</td>
</tr>
<tr>
<td>display doptions</td>
<td>14</td>
</tr>
<tr>
<td>display time</td>
<td>88</td>
</tr>
<tr>
<td>echo</td>
<td>33, 43</td>
</tr>
<tr>
<td>enter</td>
<td>48</td>
</tr>
<tr>
<td>hardreset</td>
<td>51</td>
</tr>
<tr>
<td>help</td>
<td>5</td>
</tr>
<tr>
<td>helpall</td>
<td>5</td>
</tr>
<tr>
<td>helpexamples</td>
<td>5, 8</td>
</tr>
<tr>
<td>helpnewer</td>
<td>5, 7</td>
</tr>
<tr>
<td>helpnext</td>
<td>5, 8</td>
</tr>
<tr>
<td>helpoptions</td>
<td>5, 8</td>
</tr>
<tr>
<td>helpsubject *</td>
<td>7</td>
</tr>
<tr>
<td>helpsubject</td>
<td>5, 8, 7</td>
</tr>
<tr>
<td>helpusage</td>
<td>5, 8</td>
</tr>
<tr>
<td>cinclude_and_print</td>
<td>88</td>
</tr>
<tr>
<td>include_files</td>
<td>33</td>
</tr>
<tr>
<td>ListEdit</td>
<td>10</td>
</tr>
<tr>
<td>matop</td>
<td>50</td>
</tr>
</tbody>
</table>
MatrixFunc ................................................................. 39
memdate ................................................................. 65
pbf ................................................................. 32, 37
plot ................................................................. 18
printfancy ............................................................... 13
print_scaled ............................................................. 12
prompt ................................................................. 24
reset_gpenhdtl ......................................................... 29
store ................................................................. 54, 64
remove ................................................................. 76
restore ................................................................. 54, 64
resume ................................................................. 26
rewind ................................................................. 30
runprint ................................................................. 53
set_option ............................................................... 13
store ................................................................. 54
suspend ................................................................. 48
terminal ................................................................. 67
trace ................................................................. 65, 67, 73
user_name_is ............................................................ 70
whatis ................................................................. 43, 52
Whatis ................................................................. 52
whereis ................................................................. 52
commands, creating new ............................................. 17
compiler, what is a .................................................... 35
compiler-reported errors .............................................. 41
Convention, .................................................................
  Auto-Pushback .......................................................... 15
  Double-Quote .......................................................... 10
  <FILENAME> ............................................................ 27
  No-Quote ............................................................... 10
  </PATTERN/FILENAME> ................................................ 28
procedure naming ..................................................... 17
cpytoeof built-in routine ............................................. 48
CREATEFILE define ................................................... 30
creating commands .................................................... 17
creating new DELIGHT versions ................................... 54
data base of variable declarations ................................ 74
debugger ................................................................. 50
debugging Rattle execution ......................................... 40
decglo subroutine ..................................................... 60
deci subroutine ........................................................ 59
decia subroutine ....................................................... 59
decia subroutine ....................................................... 59
decia2 subroutine ...................................................... 59
decia3 subroutine ...................................................... 59
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>decr subroutine</td>
<td>59</td>
</tr>
<tr>
<td>decra1 subroutine</td>
<td>59</td>
</tr>
<tr>
<td>decra2 subroutine</td>
<td>59</td>
</tr>
<tr>
<td>decra3 subroutine</td>
<td>59</td>
</tr>
<tr>
<td>define enhancements</td>
<td>9</td>
</tr>
<tr>
<td>define options</td>
<td>11</td>
</tr>
<tr>
<td>DELIGHT For Beginners</td>
<td>4</td>
</tr>
<tr>
<td>DELIGHT versions</td>
<td>54</td>
</tr>
<tr>
<td>DELIGHT.MIMO</td>
<td>8, 54</td>
</tr>
<tr>
<td>DELIGHT.SPICE</td>
<td>8, 54</td>
</tr>
<tr>
<td>DELIGHT.VNAME</td>
<td>54</td>
</tr>
<tr>
<td>directory, meaning of</td>
<td>27</td>
</tr>
<tr>
<td>display options command</td>
<td>14</td>
</tr>
<tr>
<td>display_time command</td>
<td>68</td>
</tr>
<tr>
<td>divide-by-zero run-time error</td>
<td>50</td>
</tr>
<tr>
<td>DLoptions define</td>
<td>50</td>
</tr>
<tr>
<td>DLoptions option ~AbortOnOverflow</td>
<td>50</td>
</tr>
<tr>
<td>Double-Quote Convention</td>
<td>10</td>
</tr>
<tr>
<td>double-precision arguments to built-in routines</td>
<td>58</td>
</tr>
<tr>
<td>double-precision floating-point numbers</td>
<td>56</td>
</tr>
<tr>
<td>dveccs subroutine</td>
<td>58, 61</td>
</tr>
<tr>
<td>dveexit subroutine</td>
<td>61</td>
</tr>
<tr>
<td>dvinit subroutine</td>
<td>60, 61</td>
</tr>
<tr>
<td>duname subroutine</td>
<td>61</td>
</tr>
<tr>
<td>echo command</td>
<td>33, 43</td>
</tr>
<tr>
<td>echoing input lines</td>
<td>44</td>
</tr>
<tr>
<td>enter command</td>
<td>46</td>
</tr>
<tr>
<td>environments</td>
<td>52</td>
</tr>
<tr>
<td>ERROR define</td>
<td>30</td>
</tr>
<tr>
<td>escape character</td>
<td>4</td>
</tr>
<tr>
<td>&lt;Esetup&gt; file</td>
<td>87</td>
</tr>
<tr>
<td>exedit built-in routine</td>
<td>48</td>
</tr>
<tr>
<td>expression continuation</td>
<td>42</td>
</tr>
<tr>
<td>extensibility of DELIGHT</td>
<td>35</td>
</tr>
<tr>
<td>field descriptors in answer_to_prompt first argument</td>
<td>24</td>
</tr>
<tr>
<td>file input and output</td>
<td>27</td>
</tr>
<tr>
<td>files:</td>
<td></td>
</tr>
<tr>
<td>&lt;ansprompt&gt;</td>
<td>24</td>
</tr>
<tr>
<td>&lt;Esetup&gt;</td>
<td>87</td>
</tr>
<tr>
<td>&lt;HsBASIC&gt;</td>
<td>8</td>
</tr>
<tr>
<td>&lt;HsVNAME&gt;</td>
<td>82</td>
</tr>
<tr>
<td>&lt;incfiles&gt;</td>
<td>33</td>
</tr>
<tr>
<td>&lt;HmgVNAME&gt;</td>
<td>82</td>
</tr>
<tr>
<td>&lt;macdefs&gt;</td>
<td>87</td>
</tr>
<tr>
<td>&lt;memVNAME&gt;</td>
<td>82</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>&lt;Tmatfunc&gt;</code></td>
<td>File for intermediates</td>
</tr>
<tr>
<td><code>&lt;Topuniq&gt;</code></td>
<td>File for intermediates</td>
</tr>
<tr>
<td><code>&lt;vport4&gt;</code></td>
<td>File for intermediates</td>
</tr>
<tr>
<td>file, file, file</td>
<td>binary help</td>
</tr>
<tr>
<td>help</td>
<td>Help command</td>
</tr>
<tr>
<td>openhdtl</td>
<td>Open DEUGHT object file</td>
</tr>
<tr>
<td>scratch</td>
<td>scratch</td>
</tr>
<tr>
<td>temporary, scratch</td>
<td>temporary, scratch</td>
</tr>
<tr>
<td>unique, temporary, scratch</td>
<td>unique, temporary, scratch</td>
</tr>
<tr>
<td><code>&lt;FILENAME&gt;</code></td>
<td>Convention</td>
</tr>
<tr>
<td>filenames, portability considerations for</td>
<td>filenames, portability considerations for</td>
</tr>
<tr>
<td><code>filprm</code></td>
<td>Built-in routine</td>
</tr>
<tr>
<td>floating-point exceptions</td>
<td>floating-point exceptions</td>
</tr>
<tr>
<td>Fortran language</td>
<td>Fortran language</td>
</tr>
<tr>
<td><code>GET_LETTER</code></td>
<td>Define</td>
</tr>
<tr>
<td><code>GET_NAME</code></td>
<td>Define</td>
</tr>
<tr>
<td><code>GET_NUMBER</code></td>
<td>Define</td>
</tr>
<tr>
<td><code>gtoken</code></td>
<td>Built-in routine</td>
</tr>
<tr>
<td>hardreset</td>
<td>Command</td>
</tr>
<tr>
<td>help</td>
<td>Command</td>
</tr>
<tr>
<td>help file</td>
<td>Help file</td>
</tr>
<tr>
<td><code>helpall</code></td>
<td>Command</td>
</tr>
<tr>
<td>helper</td>
<td>Helper</td>
</tr>
<tr>
<td><code>helpexamples</code></td>
<td>Command</td>
</tr>
<tr>
<td><code>helpnewer</code></td>
<td>Command</td>
</tr>
<tr>
<td><code>helpnext</code></td>
<td>Command</td>
</tr>
<tr>
<td><code>helpoptions</code></td>
<td>Command</td>
</tr>
<tr>
<td><code>helpsubject</code></td>
<td>Command</td>
</tr>
<tr>
<td><code>helpusage</code></td>
<td>Command</td>
</tr>
<tr>
<td><code>&lt;HsBASIC&gt;</code></td>
<td>Helper file</td>
</tr>
<tr>
<td><code>&lt;HsVNAME&gt;</code></td>
<td>Version-specific file</td>
</tr>
<tr>
<td><code>incfiles</code></td>
<td>File</td>
</tr>
<tr>
<td><code>cinclude</code></td>
<td>And_print</td>
</tr>
<tr>
<td><code>include_files</code></td>
<td>Command</td>
</tr>
<tr>
<td><code>inverse</code></td>
<td>Keyword to MatrixFunc command</td>
</tr>
<tr>
<td><code>round</code></td>
<td>Built-in routine</td>
</tr>
<tr>
<td>lexical analyzer part of compiler</td>
<td>lexical analyzer part of compiler</td>
</tr>
<tr>
<td>libraries, DEUGHT object file</td>
<td>libraries, DEUGHT object file</td>
</tr>
<tr>
<td>Line Continuation Convention</td>
<td>Line Continuation Convention</td>
</tr>
<tr>
<td><code>ListEdit</code></td>
<td>Command</td>
</tr>
</tbody>
</table>
DEUGHT For Intermediates (3/1/85)

list option -numbers ............................................................ 46
<lmg VNAME> version-specific file ............................................ 62
load/linkage phase .............................................................. 48
load/linking DEUGHT ............................................................ 64
local procedure variables .................................................... 46
logical unit number ............................................................. 29
<macdefs> file ................................................................... 67
macros ............................................................................. 34, 37
making a new memfile ............................................................ 55, 64
matop command ................................................................ 50
MatrixFunc command ......................................................... 39
MAXREAL define ................................................................. 50
memdate command ................................................................ 85
memfiles ............................................................................. 54
memfio subroutine ................................................................ 81, 62, 72
memfio routines for reading/writing to/from a memfile: .......... 63
   rbins ........................................................................... 83
   rbinsa ........................................................................ 83
   rbinsr ........................................................................ 83
   rbinsra ...................................................................... 83
   wbinsi ........................................................................ 83
   wbinsia ..................................................................... 83
   wbinsr ....................................................................... 83
   wbinsra ..................................................................... 83
   <memVNAME> version-specific file ..................................... 62
   multiline defines .............................................................. 10
   newline character ............................................................. 4
   NEWLINE (character) define ............................................. 37
   No-Quote Convention ..................................................... 10
   numeric overflow ............................................................ 50
   object file libraries ........................................................ 64
   online help system .......................................................... 5
  openhdtl file ................................................................... 26, 51
   openp built-in routine ..................................................... 26, 28, 47, 50
   optional define arguments ............................................. 10
   opuniq built-in routine .................................................... 31
   overflow, aborting on numeric ........................................ 50
   parser .......................................................................... 35
   passing arguments to built-in routines ............................... 58
   </PATTERN/FILENAME> Convention ....................................... 28
   pbdump built-in routine ..................................................... 42
   pbf command .................................................................. 32, 37
   pdebug_variable ............................................................ 46, 58, 73
   Phase I-II-III Method of Feasible Directions ..................... 74
   PI define ........................................................................ 35
### Table of Contents

<table>
<thead>
<tr>
<th>Command/Option</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>plot command</td>
<td>18</td>
</tr>
<tr>
<td>plot options:</td>
<td></td>
</tr>
<tr>
<td>- erase</td>
<td>20</td>
</tr>
<tr>
<td>- intxlabel and intylabel</td>
<td>20</td>
</tr>
<tr>
<td>- logx</td>
<td>22</td>
</tr>
<tr>
<td>- origin</td>
<td>22</td>
</tr>
<tr>
<td>- verbose and axisfirst</td>
<td>22</td>
</tr>
<tr>
<td>- usespr</td>
<td>22</td>
</tr>
<tr>
<td>- xmin and xmax</td>
<td>20</td>
</tr>
<tr>
<td>- xorigin and yorigin</td>
<td>22</td>
</tr>
<tr>
<td>pool of nonlocal Rattle variables</td>
<td>59</td>
</tr>
<tr>
<td>position of define arguments</td>
<td>53</td>
</tr>
<tr>
<td>position of define options</td>
<td>53</td>
</tr>
<tr>
<td>Present Input</td>
<td>29</td>
</tr>
<tr>
<td>Present Output</td>
<td>29</td>
</tr>
<tr>
<td>printf built-in routine</td>
<td>47</td>
</tr>
<tr>
<td>printf-like field descriptors</td>
<td>24</td>
</tr>
<tr>
<td>printfancy command</td>
<td>13</td>
</tr>
<tr>
<td>print_scaled command</td>
<td>12</td>
</tr>
<tr>
<td>procedure naming conventions</td>
<td>17</td>
</tr>
<tr>
<td>procedures vs multilime defines</td>
<td>18</td>
</tr>
<tr>
<td>program debugger</td>
<td>50</td>
</tr>
<tr>
<td>prompt command</td>
<td>24</td>
</tr>
<tr>
<td>push-back mechanism</td>
<td>35</td>
</tr>
<tr>
<td>rbin subroutine</td>
<td>63</td>
</tr>
<tr>
<td>rbinia subroutine</td>
<td>63</td>
</tr>
<tr>
<td>rbinr subroutine</td>
<td>63</td>
</tr>
<tr>
<td>rbinra subroutine</td>
<td>63</td>
</tr>
<tr>
<td>READMODE define</td>
<td>30</td>
</tr>
<tr>
<td>reset_openhdtl command</td>
<td>29</td>
</tr>
<tr>
<td>store command</td>
<td>54, 64</td>
</tr>
<tr>
<td>rarray Fortran double-precision array</td>
<td>58</td>
</tr>
<tr>
<td>rarray(retp) for returning function values</td>
<td>56</td>
</tr>
<tr>
<td>rcopy built-in routine</td>
<td>56</td>
</tr>
<tr>
<td>remove command</td>
<td>76</td>
</tr>
<tr>
<td>restore command</td>
<td>54, 64</td>
</tr>
<tr>
<td>resume command</td>
<td>26</td>
</tr>
<tr>
<td>rewind command</td>
<td>30</td>
</tr>
<tr>
<td>richan built-in routine</td>
<td>29, 47</td>
</tr>
<tr>
<td>rochan built-in routine</td>
<td>29</td>
</tr>
<tr>
<td>RUN-TIME ERROR message</td>
<td>50</td>
</tr>
<tr>
<td>run-time errors</td>
<td>44</td>
</tr>
<tr>
<td>rurunprint command</td>
<td>53</td>
</tr>
<tr>
<td>scanner part of compiler</td>
<td>35</td>
</tr>
<tr>
<td>scratch files and routine opuniq</td>
<td>31</td>
</tr>
</tbody>
</table>
sdelay built-in routine ................................................................. 30, 48
set_option command ................................................................. 13
setup file ................................................................................... 66
sichan built-in routine ................................................................. 30, 48
simulator_flags define ................................................................. 16
sochan built-in routine ................................................................. 29
source program to a compiler ....................................................... 35
standard places for file locations .................................................. 27
starting DELIGHT ......................................................................... 70
startup file ................................................................................... 70
store command ............................................................................ 54
<subject> help entry ...................................................................... 6
suspend command .......................................................................... 48
symbol table of DELIGHT ............................................................. 52
syntax analyzer part of compiler .................................................. 35
temporary, scratch files and routine opuniq .................................. 31
terminal command ......................................................................... 67
terminal type ................................................................................ 19
<Tmatego> file ............................................................................. 40
token ............................................................................................ 35
<Topuniq> file .............................................................................. 32
trace output and debugging .......................................................... 17
trace command ............................................................................ 45, 67, 73
trace_pushback_variable .............................................................. 42
unique, temporary, scratch files and routine opuniq ...................... 31
UNIX, references to ...................................................................... 4, 28, 34, 50, 51, 52, 58
user_name.js command .................................................................. 70
variable declaration routines: ........................................................ 59
deglo ............................................................................................ 60
deci ............................................................................................... 59
decia1 .......................................................................................... 59
decia2 .......................................................................................... 59
decia3 .......................................................................................... 59
decr ............................................................................................. 59
decra1 .......................................................................................... 59
decra2 .......................................................................................... 59
decra3 .......................................................................................... 59
version-specific files: .................................................................... 62
<ImgVNAME> ................................................................................. 62
<memVNAME> .............................................................................. 62
<HisVNAME> ................................................................................. 62
version-specific routines: .............................................................. 61
dudescs ......................................................................................... 58, 61
duezi ............................................................................................. 61
dvinit ............................................................................................ 60, 61
<table>
<thead>
<tr>
<th>Command/Keyword</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>durname</td>
<td>61</td>
</tr>
<tr>
<td>memfio</td>
<td>61, 62, 72</td>
</tr>
<tr>
<td>viewport</td>
<td>20</td>
</tr>
<tr>
<td>&lt;viewport&gt; file</td>
<td>20</td>
</tr>
<tr>
<td>wbinr subroutine</td>
<td>63</td>
</tr>
<tr>
<td>wbinia subroutine</td>
<td>63</td>
</tr>
<tr>
<td>wbinr subroutine</td>
<td>63</td>
</tr>
<tr>
<td>whatis command</td>
<td>43, 52</td>
</tr>
<tr>
<td>whereis command</td>
<td>52</td>
</tr>
<tr>
<td>work work array</td>
<td>56</td>
</tr>
<tr>
<td>WRITEMODE define</td>
<td>30</td>
</tr>
<tr>
<td>xyinv routine</td>
<td>48</td>
</tr>
<tr>
<td>-echo DELIGHT option</td>
<td>71</td>
</tr>
<tr>
<td>-fix DEIGHT option</td>
<td>71, 72</td>
</tr>
<tr>
<td>-force DELIGHT option</td>
<td>66, 71</td>
</tr>
<tr>
<td>-XXXXXX DEIGHT option</td>
<td>71, 72</td>
</tr>
<tr>
<td>-makehelp DEIGHT option</td>
<td>71</td>
</tr>
<tr>
<td>-makehelp DEIGHT option</td>
<td>71</td>
</tr>
<tr>
<td>-trsrc DEIGHT option</td>
<td>71</td>
</tr>
<tr>
<td>~Abort OnOverflow DLoptions option</td>
<td>50</td>
</tr>
<tr>
<td>~erase plot option</td>
<td>20</td>
</tr>
<tr>
<td>~intxlabel and ~intylabel plot options</td>
<td>20</td>
</tr>
<tr>
<td>~logx plot option</td>
<td>22</td>
</tr>
<tr>
<td>~numbers list option</td>
<td>46</td>
</tr>
<tr>
<td>~origin plot option</td>
<td>22</td>
</tr>
<tr>
<td>~verbose and ~axis first plot options</td>
<td>22</td>
</tr>
<tr>
<td>~usepr plot option</td>
<td>22</td>
</tr>
<tr>
<td>~xmin and ~xmax plot options</td>
<td>20</td>
</tr>
<tr>
<td>~xorigin and ~yorigin plot options</td>
<td>22</td>
</tr>
<tr>
<td>%Z helper macro</td>
<td>68</td>
</tr>
</tbody>
</table>