The Basis System

Release 12.1

The Basis Development Team

November 13, 2007

Lawrence Livermore National Laboratory
Email: basis-devel@lists.llnl.gov

COPYRIGHT NOTICE

DISCLAIMER

DOE Order 1360.4A Notice

UCRL-MA-118543

CONTENTS

CHAPTER

ONE

The Basis System

1.1 Environment Variables

Before using Basis, you should set some environment variables as follows.

- `BASIS_ROOT` should contain the name of the root of your Basis installation, `/usr/apps/basis` for example.
- `MANPATH` should contain a component `$BASIS_ROOT/man`.
- Your path should contain a component `$BASIS_ROOT/bin`.
- `DISPLAY` should contain the name of your X-Windows display, if you will be doing X-window plotting.
- `NCARG_ROOT` should contain the name of the root directory of your NCAR 4.0.1 or later distribution, if you have it.

Check with your System Manager for the exact specifications on your local systems.

1.2 Basis Is Both a Program and a Development System

Basis is a system for developing interactive computer programs in Fortran, with some support for C and C++ as well. Using Basis you can create a program that has a sophisticated programming language as its user interface so that the user can set, calculate with, and plot, all the major variables in the program. The program author writes only the scientific part of the program; Basis supplies an environment in which to exercise that scientific programming, which includes an interactive language, an interpreter, graphics, terminal logs, error recovery, macros, saving and retrieving variables, formatted I/O, and on-line documentation.

`basis` is the name of the program which results from loading the Basis System with no attached physics. It is a useful program for interactive calculations and graphics. Authors create other programs by specifying one or more packages of variables and modules to be loaded. A package

is specified using a Fortran source and a variable description file in which the user specifies the common blocks to be used in the Fortran source and the functions or subroutines that are to be callable from the interactive language parser.

Basis programs are *steerable applications*, that is, applications whose behavior can be greatly modified by their users. Basis also contains optional facilities to help authors do their jobs more easily. A library of Basis packages is available that can be added to a program in a few seconds. The programmable nature of the application simplifies testing and debugging.

The Basis Language includes variable and function declarations, graphics, several looping and conditional control structures, array syntax, operators for multiplication, dot product, transpose, array or character concatenation, and a stream I/O facility. Data types include real, double, integer, complex, logical, character, chameleon, and structure. There are more than 100 built-in functions, including all the Fortran intrinsics.

Basis' interaction with compiled routines is particularly powerful. When calling a compiled routine from the interactive language, Basis verifies the number of arguments and coerces the types of the actual arguments to match those expected by the function. A compiled function can also call a user-defined function passing arguments through common.

1.3 About This Manual

The Basis manual is presented in several parts:

I. Running a Basis Program, A Tutorial

II. Basis Language Reference

III. EZN User Manual: The Basis Graphics Package

IV. The EZD Interface

V. Writing Basis Programs: A Manual For Program Authors

VI. The Basis Package Library

VII. MPPL Reference Manual

The first three parts form a basic document set for a user of programs written with Basis. The remainder form a document set for an author of such programs.

Basis is available on most Unix and Unix-variant platforms. It is not available for Windows or Macintosh operating systems.

A great many people have helped create Basis and its documentation. The original author was Paul Dubois. Other major contributors, in alphabetical order, have been Robyn Allsman, Kelly Barrett, Cathleen Benedetti, Stewart Brown, Lee Busby, Yu-Hsing Chiu, Jim Crotinger, Barbara Dubois, Fred Fritsch, David Kershaw, Bruce Langdon, Zane Motteler, Jeff Painter, David Sinck,

Allan Springer, Bert Still, Janet Takemoto, Lee Taylor, Susan Taylor, Peter Willmann, and Sharon Wilson. The authors of this manual stand as representative of their efforts and those of a much larger number of additional contributors.

Send any comments about these documents to ”basis-devel@lists.llnl.gov” on the Internet.

Part I

Running a Basis Program, A Tutorial

CHAPTER

TWO

Getting Started

2.1 What is Basis?

Basis is two things: It is a system used to produce computer programs, and it is the name of a programming language which serves as the user interface to a program so produced. To say it another way, an author uses the Basis System to make a program named **foo**, and the user uses the Basis Language to write the input file for **foo**. Just to add to the confusion, one such **foo** is a program named **basis**, which consists of nothing but an interpreter for the Basis Language.

The purpose of this manual is to give you a quick introduction to working with the Basis Language, so that you can get going as rapidly as possible. This tutorial is not a complete description of the Basis Language, but is intended to build enough of a foundation that you can later learn more sophisticated features from the full reference manual. From now on, when we say Basis, we mean the Basis Language, not the Basis System that the author used to build **foo**. When we say **basis**, we mean the program basis which you can use as a practice vehicle for learning the language, or as a useful interactive calculator and plotter.

The Basis Language looks very much like Fortran, so if you know Fortran, you should be able to pick up the elements very quickly. Unlike Fortran, though, Basis is an interpreted language, which means that (usually) Basis statements are executed as soon as they are typed in. Basis contains a lot of built-in functions, input-output facilities, and can interact with compiled code and variables. You can even compute Basis' input with Basis itself.

2.2 Starting the Program

If you are using basis by itself, you simply type in

```
basis
```

on the computer of your choice. You may be using some other code (Lasnex is an example) which consists of Basis and a lot of other compiled code, in which case you may type in some other name.

Basis (or whatever) will initialize and when the process is complete, it will print out a prompt for you. The usual default prompt is

```
Basis>
```

although this is one of the many customizable features of Basis that can be changed by the program author. For sure, though, you will know when you are being prompted. Upon receiving the prompt, you can immediately begin typing in Basis statements.

The default for a program built with the Basis System is that any arguments on the command line are simply treated as the first line of input. However, this is one of those things an author may have changed, so check your specific program's documentation.

2.3 Getting Information

The key command for finding your way around a Basis program is the `LIST` command. Basis commands such as `LIST` may be entered either in all lower case or all upper case. Sometimes we will use upper case to emphasize that the word is a Basis reserved word, but usually people enter them in lower case.

If you enter the command

```
list
```

you will get output something like the following:

```
list options
------------
list                    [print the list options]
list par.Attributes
list [pkg.]functions
list Groupname
list [pkg.]groups
list idname
list macros
list packages
list [pkg.]variables
NOTE:Groupname is the name of a group in any package on the
     current search stack.
NOTE:Groupname can be abbreviated.
NOTE:idname is the name of a function, macro, or variable in any
     package on the current search stack.
NOTE:list groups, functions, and variables list local and user
     created groups, functions and variables respectively
     unless pkg. is utilized.
NOTE:list pkg.functions lists the built-in and compiled functions
     in the database for that package.
```

You can use the `LIST` command to get all sorts of information about Basis functions, predefined macros, constants and variables, and the like. Enter:

```
list packages
```

and Basis will list the packages that are loaded. One of these is *par*, the Basis parser package. Now enter

```
list par.groups
```

and you will see a list of the groups that make up this package. Now you can enter `LIST` followed by one of the group names (or a prefix of it) and you will see an explanatory list of the items in that group. You can ask to have an individual item listed (such as a compiled or built-in function) to get more information about that item (for instance, its parameters, what it does, what type it returns (if any), and so on).

2.4 Comparison of Basis and Fortran

We summarize here very briefly the important similarities and differences between Basis and Fortran. In each section of this manual we will summarize further similarities or differences pertinent to the topic under discussion. For the real details you will need to go to the reference manual.

2.4.1 Major Similarities between Basis and Fortran

1. Basis has pretty much all of the Fortran operators and delimiters you are familiar with, and they have the same functions and precedences. It has many more operators, but you won't need them when you're getting started.
2. Basis expressions (including array references and function invocations) look just like they do in Fortran.
3. Basis has all the data types available in Fortran (and more).
4. Basis `IF` statements look just like Fortran `IF` statements.
5. Basis `DO` statements are very similar to Fortran `DO` statements, but have no label and end in an `ENDDO` statement.

2.4.2 Major Differences between Basis and Fortran

Mostly, Basis will do what an experienced Fortran programmer expects. The chief incompatibility is in the form of the input. In general, Basis extends the ideas of Fortran to an array-syntax interpretive environment.

1. Basis is interpreted rather than compiled.
2. Basis comments start with # and extend to the end of the line.
3. Basis has no statement numbers or goto's.
4. Basis input is essentially free-form (columns are not significant). There is no continuation column; a statement is continued from one line to the next by ending the line with or a comma, open paren or bracket, or operator.
5. There are no default types in Basis; variables must be declared.
6. Basis function names and formal arguments must NOT be typed.
7. Basis functions may return virtually any kind of entity, including arrays.
8. Basis passes actual parameters to functions by value (i.e., as copies), not by reference (i.e., as addresses).
9. Several Basis statements can appear on the same line, if separated by semicolons(;).
10. Basis is case sensitive, that is, upper and lower case are distinguished. Basis reserved words may be entered either in all caps or all lower case, though.
11. Spaces are significant in Basis (except in quoted strings and comments, of course), and act as delimiters between tokens.
12. Double quotes are used for strings – single quotes are used for something else.

CHAPTER

THREE

The Basis Language

3.1 Assignments and Expressions

One of the first things you are going to want to do is assign values to variables. These may be variables which are already built into Basis, but could also be variables in compiled code which the author has made available to the Basis. In a future chapter, we'll show you how to declare your own Basis variables, and then, of course, you can assign values to these as well. All variables to which some quantity is assigned must have been declared previously, either by Basis, by you, or by the program author.

Here are some examples of assignments. These are all assignments to variables which are predeclared in Basis. These examples are designed for you to follow along with on a terminal.

```
debug = yes
fuzz = 9
switches(2) = pi
switches(5) = 2.0 * cos ( switches(2) / 3 )
```

yes is a predefined constant in Basis (value 1), and *pi* is a predefined famous number. *debug* is a predeclared variable which, if set to *yes*, toggles on more detailed debugging diagnostics. We recommend '`debug=yes`' for beginners. *fuzz* controls the accuracy of real numbers printed out by Basis (it is the number of digits after the decimal point). *switches* is a predeclared real scratch array which we have used here to show assignment to an array element. The last assignment illustrates an arithmetic statement used in an assignment; its meaning should be obvious to any user of Fortran. `cos` is only one of a very large number of built-in functions available in Basis. All the usual ones are available; see the reference manual for complete details, or use the `LIST` command to find out more.

For your convenience, Basis has predefined the following constants: *yes = 1*, *no = 0*, *on = "on"*, *off = "off"*, *true = logical true*, and *false = logical false*. In many cases `true` and `yes` can be used interchangeably, as can `false` and `no`.

If you want to print out the values of some of the above variables, simply type in a comma-delimited list of their names, for example

```
fuzz , switches(2)
```

and Basis will print out the values (if real, to the number of digits specified by *fuzz*). In fact, you can type in any expression or list of expressions, and Basis will compute it and print out its value (if it can). For instance, try typing in

```
2.0 * cos ( switches(2) / 3 ), x + y + 5
```

You should get an "Unknown variable" message from the second expression. When Basis encounters an error, it stops processing the current line, writes a diagnostic to the terminal and some debugging information to a trace file (more if *debug* is *yes*, as mentioned above), and returns to the prompt. It is now ready to accept further statements.

Try typing in some sort of declaration for *x* and *y*, such as

```
integer x , y
```

and then assign something to them, then type the expression again. This time you will get its computed value. At this point, assuming you know Fortran, you should be able to type in various Fortran-compatible declarations, expressions, and assignment statements, to get more of a feel for Basis. Don't worry about making mistakes; Basis is very tolerant of them and will come back again and again.

You might also want to experiment with the predeclared Basis chameleon variables *$a*, *$b*, ... , *$z*. These variables exist in Basis but initially have no type or value; they get their types and values by being assigned to, and this can be done again and again. They are thus called chameleons for the obvious reason that they "change color" to "blend into their environment", i.e., change type and value to whatever is assigned to them. Try the following sequence of statements:

```
$b
$b = cos ( pi / 3.0 )
$b
$b = 6
$b
```

The first statement will cause an error because, although *$b* exists, it is undefined. The second statement assigns *$b* a real value, which the third prints. The fourth assigns it an integer value, which the fifth prints. You might want to experiment a bit with the chameleons before proceeding.

Basis allows logical variables and logical-valued expressions. As in Fortran, such expressions would normally be used in an `IF` statement to control some execution choice. However, logical values can be assigned too.

```
integer x, y, m
x = 3
```

```
y = 5
logical z
z = x > y
z
```

Basis will tell you that the value of *z* is `false`. You can use either $>$ or the more Fortran-like `.gt.` for the comparison operator. If you do use `.gt.` make sure the periods are not ambiguous. For example, '`3.gt.y`' is not going to work right but '`3 .gt.y`' is ok.

3.2 Input from a File

So far we've discussed input from the terminal, at the Basis prompt. Often, however, we input statements from a file. If you have code in a file named '`my_funcs`' then you can have it read in and interpreted by entering:

```
read my_funcs # no quotes necessary here
```

A file that you are reading in can itself contain `read` commands, and so on, up to a depth of twenty. If Basis detects an error in a file that it is reading in, it will close the file and give a diagnostic, then return to the Basis prompt for input. A `resume` statement will begin reading that file again at the line that failed (assuming you snuck off and fixed it, and the line was at an appropriate place to resume input).

You can execute commands using the Bourne shell from within Basis (like starting up your editor in order to fix the input file) by beginning the line with an exclamation point:

```
!emacs my_funcs
```

You will return to Basis when the command exits.

3.3 Some Differences from Fortran

There is a lot more to Basis expressions than just imitating Fortran. Basis operators are more general, and there are more of them.

1. Most Basis operators are more general than their Fortran counterparts. For instance, '`*`' and other arithmetic operators will perform component-wise operations on vector or array arguments.

2. Basis has many additional operators such as matrix multiply and dot product; the operator '`//`' which concatenates strings and arrays.

3. Unlike Fortran, the Basis logical and relational operators also have symbolic versions; for instance '`&`' for `.and.`; '`~=`' or '`<>`' for '`.ne.`', and the like.

We'll cover more of this as we proceed.

3.4 Declaring Variables

The examples in this chapter are designed to be executed as you read them. For brevity, variables declared in an earlier example are frequently reused without redeclaring them in later examples.

The usual Fortran types are available for declaring variables, such as:

```
INTEGER x, y, z
REAL i, j, k = 2.0
DOUBLE d = 2.d0
COMPLEX c = 2.0 + 3.0i
LOGICAL l1 = true, l2 = false
CHARACTER*3 ch = "abc"
```

Note the following (mostly minor) differences from Fortran:

- Variables can be initialized in their declarations (as `k`, `d`, `c`, `l1`, `l2`, and `ch` above). These initialization expressions need not be just constants, but can be arbitrary expressions, as long as all values in them are known when the statement is encountered. Even '`real x=1, y=x`' is ok.
- The use of `DOUBLE` alone, not `DOUBLE PRECISION`.
- The notation for imaginary constants (a numerical quantity followed by `i`, with no space between).
- `true` and `false` without the surrounding periods as in Fortran.
- Variables which are not explicitly initialized are set to 0, or to blanks if they are of character type.

Declare array variables of up to seven dimensions as follows:

```
REAL x(10), y(3,5), z(-3:5, 7:10)
```

The lowest value of the subscript range defaults to 1 unless a different value is specified before a colon, as in `z` above. Thus, `x` is subscripted $1 \ldots 10$, `y` from $1 \ldots 3$ and $1 \ldots 5$, and `z` from $-3 \ldots 5$ and $7 \ldots 10$. An individual array can be initialized by a vector of values that follows its type declaration:

```
INTEGER i(10) = [0,0,0,0,0,1,1,1,1,1], j(5) = [1,2,3,4,5]
```

The vector components may be arbitrary expressions.

3.5 Some Elements of Array Syntax

Basis operators support arithmetic on arrays of arbitrary size and shape, except that binary operations (`*`, `/`, etc.) require their operands to be compatible in size and shape; and assignments require that the object being assigned must be storable as a subobject of the receiving item. For all the details on this subject the reader is referred to the Basis reference manual.

Array operands can be expressed in various ways. An entire array is specified by its name without subscripts: Thus, in the example above, either `i` or `i()` refers to the entire array of ten elements as specified. One can also use subscript range notation to extract subarrays of a given array.

```
i(1:5) # will be the first five elements of i
i(2:10:2) # will be [i(2), i(4), i(6), i(8), i(10)]
i(3:5)+j(1:3) # will be [1,2,3]
```

A range specification consists of

```
low_dimension:high_dimension:step_size
```

`step_size`, if omitted, defaults to 1. `low_dimension` and `high_dimension` must be within the declared range, and if either is omitted, defaults to the declared value. Aside: These numbers must be integers. The range notation with a real component means a vector of real numbers. Try entering

```
0.:1.:10   #vector of ten reals from 0. to 1.
0.:1.:.02  #From 0. to 1. in steps of .02
```

By contrast, if you enter `0:10:2` you are printing out a range and such a range can be used as a subscript.

You can apply the usual binary operators to objects of the same size and shape (regardless of subscript values), and the operation will be applied to each component. The only exception to this compatibility requirement is that scalars may participate in operations with arrays, in which case the scalar is applied to each element of the array. For example,

```
i + 5 # adds 5 to each component of i
```

The same rules apply to arrays with more than one dimension. There is an additional rule applying when subscripts are missing. A missing subscript will always default to its minimum value (as declared), except when all are missing, which means the entire array. Examples:

```
integer a(5,5)
a()          # is the entire 5 by 5 array
a(5)         # is just a(5,1)
a(1:5)       # is [a(1,1),a(2,1),a(3,1),a(4,1),a(5,1)]
a(1:5) + j   # valid operation since j is the same size and shape
```

3.6 IF Statements

Basis `IF` statements are exactly like Fortran, except that it is possible (and preferable, we think) to use these comparison operators:

- > instead of `.gt.`
- >= instead of `.ge.`
- < instead of `.lt.`
- <= instead of `.le.`
- = or == instead of `.eq.`
- <> or ~= instead of `.ne.`
- ~ instead of `.not.`

The following will determine the maximum of two numbers:

```
 if ( x > y ) then
   m = x
 else
   m = y
endif
```

Unlike Fortran, you do not use variant forms of the comparison operators for non-numeric types. Most Fortran programmers are blissfully unaware of `.eqv.`, but if you know about it, forget it.

3.7 Looping Constructs

Basis has several looping constructs. The most-used one is a `DO/ENDDO` statement that is close to the Fortran `DO`.

Suppose, for example, that *a*, *b*, and *c* are all *n* by *n* square matrices, and that we want to put the matrix product of *b* and *c* into *a*. This could be done by the following:

```
integer i1, i2, i3, n = 5
integer a(n,n), b(n,n), c(n,n)
b = b + 1
c = c + 2      # setting values for b and c.
do i1 = 1 , n
   do i2 = 1 , n
      a(i1,i2) = 0
      do i3 = 1 , n
         a(i1,i2) = a(i1,i2) + b(i1,i3) * c(i3,i2)
      enddo
   enddo
enddo
```

The only real difference between this statement and the Fortran `DO` is that Basis does not have statement labels so the do-loops are delineated by the `DO` ... `ENDDO` pair. Since there is no statement number, none appears after the reserved word `DO`. As in Fortran, an increment can be specified, but if not, it defaults to 1.

3.8 Vector Syntax

We can greatly increase the speed of array calculations by using array syntax where possible. We can rewrite the matrix multiply as:

```
do i1 = 1 , n
   do i2 = 1 , n
      a(i1,i2) = sum(b(i1,) * c(,i2))
   enddo
enddo
```

or use the dot-product operator `!`:

```
do i1 = 1 , n
   do i2 = 1 , n
      a(i1,i2) = b(i1,) !c(,i2)
   enddo
enddo
```

To really make it easy, use the matrix-multiply operator `*!`:

```
a = b  *! c
```

Other matrix facilities include `transpose(a)`, and concatenation (`//`). The latter operation appends one array to the end of another, forming a one-dimensional object whose size is the total number of elements of the two components. With the preceding declarations of *i* and *j*, you might want to try

```
i//j
i + j // j
```

to see what happens, and see if you understand why.

Square brackets are used for array building notation. Up to now, we have simply shown them used for literal arrays, but you might want to experiment with them to see what they can do:

```
[[1,2],[3,4]]  # The matrix
               # 1 3
               # 2 4
[j,j+1]        #  = [[1,2,3,4,5],[2,3,4,5,6]]
[j,2,3]        # [1,2,3,4,5,2,3]
[[j,j+1],99]   # [1,2,3,4,5,2,3,4,5,6,99]
[[j,j+1],[j-1]]# [[1,2,3,4,5],[2,3,4,5,6],[0,1,2,3,4]]
```

The final tool for building arrays is the `:=` assignment operator, which appends the right hand side to the left hand size, thus changing its size. This is usually used to build up a list whose length is not known in advance.

```
integer mylist(1:0) #empty integer list
integer k
do k = 1, 100
   if(mod(k**2,6)==0) then
      mylist  := k
   endif
enddo
mylist
```

prints the list of those integers between 1 and 100 whose squares are divisible by 6.

3.9 Differences between Basis and Fortran

1. Basis does not have `DIMENSION` or `EQUIVALENCE` statements.
2. The Basis `CHARACTER` type does not allow the syntax '`character x*3 , y*19`'.
3. Basis words such as `INTEGER` (and other type names), `IF`, `DO`, etc., are reserved words and cannot be used as variable names.

4. Basis has additional types (`RANGE`, `INDIRECT`, and `CHAMELEON`), which are not available in Fortran.

5. In Basis, a type can be prefaced with a scope, such as a package name. The most frequently used of these is `GLOBAL`, as in '`global real x`'. This declaration makes *x* a global variable and therefore *x* will exist even after the return of the function in which this declaration occurs.

6. Most Basis functions (e.g., `sqrt`, `sin`,`cos`, `exp`, etc.) will accept arrays as arguments, perform the indicated function on components, and return an array of the results.

7. Basis has many more types of looping statements, such as `DO`... `UNTIL`, `FOR` (similar to the C statement), `WHILE`... `ENDWHILE`, etc., described in more detail in the reference manual.

CHAPTER

FOUR

Graphics

A Basis program may or may not have a graphics package attached. The standard package attached to **basis** is called `ezc`. The current version of `ezc` uses NCAR graphics and is sometimes referred to as `ezn` to distinguish it from an earlier, non-NCAR, version.

The graphics devices available depend on those available in the graphical kernel system (GKS) available at your site. At a minimum, this is NCAR's GKS, which produces output files called NCGM files, which can be processed by NCAR utilities **ctrans** and **idt**. Another GKS is one made by a company called ATC. The ATC-GKS has drivers for X-Windows, Postscript, Tektronics, and CGM files. These CGM files can be converted to NCGM files using the NCAR utility **cgm2ncgm**.

We will not attempt to reproduce the EZC manual here, but the following sample session may be enough to get you going.

Before using a program containing EZN, make sure you have set the Basis Environment Variables as described in the first chapter.

It assumes ATC-GKS and begins with turning on both CGM file output and an X-Window. It then plots two curves on the same graph, advances the frame, and makes a contour plot. For the contour plot, titles are added and the frame limits are controlled by the user.

```
abics[1] basis
Basis    (basis, Version 931116)
Run at 10:55:17 on 11/22/93 on the sun4 machine, suffix 18021x
Initializing Basis System
Basis 9a
Initializing PFB Interface
PFB 1.0
Initializing 3-D Surface Plotting Routine
Initializing Device Package
EZD Graphics Devices 2.1
Initializing EZCURVE/NCAR Graphics
ezn /NCAR/ATC 4.2
Basis> real x=iota(100),y1=x**2,y2=x**2.1
Basis> real xx=iota(-5:5),yy=xx+6,zz=outer(xx,yy)
Basis> ezcshow=false   #see below
```

```
Basis> cgm on
Beginning CGM File problem.001.cgm
Beginning CGM Log problem.001.cgmlog
Basis> win on
Basis> plot y1 x
Basis> plot y2 x color=red style=dashed
Basis> nf
Basis> frame -4. 4. 0. 10.
Basis> titles "Top" "Bottom" "Left" "Right"
Basis> plotz zz xx yy
Basis> end
Closed CGM File problem.001.cgm,     1 frames.
Closed CGM Log File problem.001.cgmlog
   CPU (sec)   SYS (sec)
       2.733       3.000
abics[2]  cgm2ncgm < problem.001.cgm > foo.ncgm
abics[3]  ctrans -d ps.mono foo.ncgm | lpr
```

In the last two lines, the ATC-GKS CGM file was converted to an NCGM file, and the **ctrans** utility was used to send the picture to a monochrome postscript printer. To view the file in an X-Window do

```
ctrans -d X11 foo.ncgm
```

A window will appear; click in it to see the next frame.

The '`ezcshow=false`' line causes the plots to not be displayed until all the objects have been added to them and the `nf` ("new frame") is executed. Without it three frames would have been generated because the first one would have contained the plot of *y1* and the second the plot of both curves.

'`outer(xx,yy)`' forms the outer product of the vectors *xx* and *yy*, making *zz* a matrix.

Use the `LIST` command on the `ezc` package to get more ideas about what you can control.

CHAPTER

FIVE

Text Input and Output

5.1 Stream Input

5.1.1 The >> Operator

This section explains how to read numbers in from a text file, which may contain numbers in various formats as well as various non-numeric information which is to be skipped over.

The operator >> is called the "stream input" operator and it is inspired by the operator in the language C++. It has the basic form:

```
unit >> variable
```

where `unit` is an integer which has been set as the result of calling the function `basopen("filename","r")`. The 'r' stands for "read". The function `basclose(unit)` is used to close the input file when finished.

Suppose the input file 'testdata' looks like this:

```
c special input file
     time = 2.56 , factor = 13.51e-2
     1.2 2.3 3.4 4.5
```

Then here is some Basis code which would read in the numbers in this file:

```
integer i1 =  basopen("testdata","r")
real x,y, d(2,2)
i1 >> x
i1 >> y
i1 >> d
call  basclose(i1)
```

Then after the execution of the above sequence of instructions, `x=2.56`,`y = .1351`, `d(1,1) = 1.2`, `d(2,1) = 2.3`, `d(1,2) = 3.4`, and `d(2,2) = 4.5`. The remaining characters in the file (the "noise") will have been ignored. Note that *d* appeared in the input list with no subscripts (thus implying the entire array), and that it was read in in column major order (first subscript varying most rapidly). Basis is like Fortran, which also stores its arrays in column major order.

You should close the input file with:

```
call  basclose (unit)
```

NOTE: Files opened by `basopen` are automatically closed whenever an error occurs.

A series of stream input statements can be abbreviated by multiple stream input operators per statement. The above is equivalent to:

```
i1 >> x >> y >> d
```

You may use the terminal as an input file (but don't open or close it, please!) by using `stdin` as the unit number, or omitting the unit number.

5.1.2 Detecting end-of-file

It is the user's responsibility to determine whether the end of a file has been reached. For this reason an end-of-file flag (*eof*) has been provided. *eof* is an integer variable which contains the value `no` if the last read attempt was successful, and `yes` if the last read attempt was unsuccessful. The user should use the *eof* variable when reading input. For example, this is how one could read an array of unknown length (first create a file 'numbs' with some numbers in it):

```
real x(1:0), y    # x starts empty
integer i1 =  basopen("numbs", "r")
eof = no          # making sure eof is no to start with
i1 >> y           # read y
while ( eof = no )
  x  := y          # append y to x
  i1 >> y
endwhile
call  basclose(i1)
```

When the end of a file is encountered, the variables that cannot be assigned new values because of lack of input retain their original values. Once `eof` is `yes` for a specific file, the user should make no further attempt to read input from that file.

`eof` always reflects the status of the last file read from. Test its status on a particular file before you issue an input command for some other file, which may change its status.

5.2 Stream Output

5.2.1 The << Operator

Stream output is very similar to stream input, which we studied in an earlier chapter. You open the file for writing:

```
unit =  basopen ( "file" , "w" )
```

You give one or more output commands, unit number first, then an expression, and as many more operator-expression pairs as desired:

```
unit << fee << fie << fo << fum
```

Output expressions can be any legal Basis expression. Each output command will start on a new line, but may or may not be more than one line long. When finished, close the file (Note this is the same call to close an input file):

```
call  basclose ( unit )
```

You won't get any spaces between the different parts of the output unless you put them there, as in

```
unit << "x is " << x << " and y is " << y
```

You may use the terminal as an output device (but please don't open or close it!) by using `stdout` as the unit number, or by omitting the unit number. This makes it easy to make comments:

```
<< "Dear Sir, your run is proceeding quite nicely."
```

You may put stream output onto your current graphics devices by using `stdplot` as a unit number; again, neither open or close stdplot. Many Basis users like to document their graphics files by using stdplot to print the values of input parameters at the start of their graphics files. You can also redirect most terminal output to the graphics files with

```
output graphics
```

with a subsequent `output tty` to restore terminal output.

5.2.2 Controlling Line Length

You can force a line break anywhere in the output by placing the reserved word `return` between any two output operators. You can cause the automatic line break after each output command to be suppressed by setting the Basis variable `autocr` to `no` (its default value is `yes`). In this case, Basis will fill an output buffer before it sends the output and a line break, unless there is a `return` somewhere along the way. (You, the user, can do nothing to alter the size of the Basis output buffer.)

5.2.3 Formatting: format

Basis contains a built-in function `format` which takes a number and some integer parameters and returns a character string. It can be used to produce output similar to Fortran formatted output. However, `format` only accepts a scalar argument, so if you want to send out an array, it will have to be in a loop where you send elements one at a time. To format an integer, use

```
format ( <integer expression> , <field width> )
```

after an output operator. This function call returns the acsii character string for the integer expression, with exactly the number of characters asked for (right justified, if necessary), except that if you specify 0, it will give you exactly as many as are necessary.

The `format` function does not accept complex numbers, by the way, so you would have to format the real and imaginary parts separately; use `float(c)` and `cmplx(c)` to extract the real and imaginary parts.

To format a real expression, use

```
format (<expression>,<field width>,<dec. places>,<EorF>)
```

after an output operator. The string returned by this call will have exactly the number of characters specified by the width, unless you ask for 0, in which case it will give you only as many as are necessary. The number will be right justified if necessary. `<dec. places>` tells how many digits you want to the right of the decimal point. If `<EorF>` is 0, it will give a Fortran `E`-type format, and if it is 1, it will give a Fortran `F`-type format.

CHAPTER

SIX

Functions

6.1 Defining Functions

Now we'll learn how to define functions in Basis. When a function is defined, it is compiled into an internal form and stored. The function will then be executed if the function is invoked in a Basis expression.

In this section, we will look at examples of functions. By the time you finish this tutorial, you should be able to write many useful functions.

The following function computes the absolute value of the difference of its arguments.

```
FUNCTION adiff(x,y)
return abs(x-y)
ENDF
```

To try it, enter `adiff(-5.,5)`. Then try `adiff( [1,2],[9,2])`.

```
list adiff
```

displays the information that Basis has stored about this function; if you answer "`y`" to the "`Dump intermediate code?`" question, you will get a hint of what the internal code of this function looks like.

Here are some notes about the function `adiff`:

1. Note that neither the function nor its formal parameters is typed. This is what permits `adiff` to return different types and shapes of results depending on the input.
2. In Basis, a value is returned from a function by using the `RETURN` statement followed by a value (similar to C), *not* by assigning a value to the function name (as Fortran would do it).
3. The function ends with reserved word `ENDF`, not `END`. The reserved word `END`, in Basis, causes Basis to terminate. It can not be legally used in any other context.

You can declare local variables inside functions, which will not exist after they return.

```
function diff(x)
# return first differences of x
chameleon z=shape(x,length(x))
return z(2:)-z(1:length(z)-1)
endf
```

Note the use of the chameleon type; this means `diff` works properly whether `x` is integer, real, double, or complex. The shape function makes sure `z` is a vector whose lowest index is 1 so that we can subscript it correctly in the following line.

6.2 Arguments Passed by Value

Basis passes a copy of each argument to the function ("pass by value") while Fortran passes the address of the argument ("pass by reference"). Thus a Fortran function which assigns a value to one of its arguments will cause a change in the value of the actual argument, while this will not occur in Basis, since only a copy is altered.

When you call a Fortran function from Basis, and the function changes one of its arguments, you must tell Basis to pass an argument by address by prefixing the name of the argument with an ampersand:

```
real x
call second(&x)
```

Here, `second` is a compiled function which returns the time used in its argument.

There are very few cases where it is necessary to have a Basis Language function change an argument, because a function can return an entire array as its value, if necessary. However, Basis has an `INDIRECT` type that allows you to pass the name of the argument and then operate on that in the function:

```
FUNCTION w(namex)
INDIRECT y=namex
y(3) = 7.
ENDF
REAL x(100)
call w("x")
```

will result in `x(3)` being set to 7. By contrast,

```
FUNCTION w(y)
y(3) = 7.   #THIS IS USELESS
ENDF
REAL x(100)
call w(x)
```

does NOT modify `x`; rather, a copy of `x` has been modified, and then discarded when `w` returned.

In other words, the name of an argument whose value is to be changed should be passed to the function as a character string. Within the function, a local variable is declared `INDIRECT` and initialized to the name of the formal parameter to be changed. Then assignment to the `INDIRECT` variable will result in changing the actual argument in the calling routine.

6.3 Further Differences with Fortran

1. Basis does not have the `SUBROUTINE` declaration; a Basis function can return a value or not.

2. `COMMON` variables do not exist in Basis.

3. Globally accessible variables can be declared inside a Basis function (by prefacing their declaration by the additional reserved word `GLOBAL`). A global variable, that is, one declared outside of any function, is visible from any function. A local variable declared in a function is visible only within that function, where it hides a global variable of the same name.

CHAPTER

SEVEN

Built-in and Compiled Functions

There are three types of functions in Basis. The first, Basis language functions, are also called "user" functions. The other kinds are "built-in" and "compiled." Built-in functions are a special form of compiled function, either supplied as part of the Basis parser itself like `cos` or `iota` or `sqrt`, or written by a particularly Basis-skilled code developer. Compiled functions are ordinary Fortran routines whose calling sequence has been "taught" to the Basis interpreter.

Built-ins are usually used when it is desired to accept different kinds of Basis objects as arguments and return whatever type of object is appropriate. For example, many numerical-valued built-in functions will accept an arbitrary array of numbers and return an array of the same type, size and shape, whose entries were obtained by applying the function to each entry in the original array. Many of the most useful functions described below are designed to operate specifically on arrays.

For information on any individual function you can use `LIST` followed by the name of the particular function. In addition to giving you more information about what the function does, the output will also tell you what (if anything) the function returns, how many arguments it has, what their types are, etc. The Basis reference document also has more complete information. We discuss some of the more useful built-in and compiled functions in the following sections.

The *size* or *length* of an array is its total number of elements. The *shape* of an array is a vector whose components tell how many values the respective subscripts of the array can take on.

In Basis arithmetic, arrays must be of the same size and shape to participate in componentwise binary operations such as `+`, `-`, `*`, and `/`. The only exception is that one operand can be an array and the other a scalar, in which case the scalar is *broadcast*, which means that the operation is applied to the scalar versus every element of the array. Another way of thinking of it is that the scalar is expanded into an array of the same size and shape as the other operand, each element of which has the original scalar value.

Most functions which accept array arguments will also accept scalar arguments along with arrays, in which case they broadcast the scalars as described above.

Some of the functions below which accept array arguments don't care about shape, but only size. In this case they operate on corresponding components of their respective arguments, but you need to know what "corresponding components" are when the subscripts have different ranges. This is done in a fairly natural way: arrays in Basis, as Fortran, are stored in column major order (i.e., the first subscript varies most rapidly as we go through the array in memory). The elements of two

arrays of different shape but the same size are said to *correspond* if they occupy the same relative position in this memory hierarchy.

When a function is called with an argument of the wrong type, or an expression involves mixed numerical types, Basis will perform automatic type coercion if necessary. For example,

```
3+4.
```

results in the "3" being converted to "3.0" before the addition operation, and if `f(x)` is a compiled function expecting a real argument `x`, then

```
f(3)
```

actually results in

```
f(3.0)
```

In an array of numerical constants of mixed types, its elements will be coerced to the highest type in the hierarchy integer → real→ double→ complex.

Likewise, most built-in functions try to do the right thing. For example, `sqrt(2)` means `sqrt(2.0)`.

Doubles are coerced to reals, and thence to integers, by truncation. Logical `true` converts to and from integer 1, and logical `false` converts to and from integer 0.

7.1 max and min Versus sup and inf

The functions `max` and `min` accept any number of arguments. If all arguments are scalar, then the result is the largest (resp. smallest) scalar in the list. If any argument is an array, then all other arguments must be either scalar or arrays with the same number of elements (but not necessarily the same shape). The scalar arguments, if any, will be expanded to vectors of this same length, with all entries equal to the scalar. Then the maximum (or minimum) will be taken component-by-component and returned as an array. The shape of this array will be the same as the shape of the first of the original arguments that was not a scalar.

In contrast, the functions `sup` and `inf` accept any number of arguments (even a single one), either scalar or array, of arbitrary size and shape, and return the scalar value of the largest (resp. smallest) component of all the arguments. Thus these functions always return a scalar; `max` and `min` will return an array if they have any argument which is an array.

A `max` or `min` of a single argument is treated as a `sup` or `inf`, since that is what you probably meant.

7.2 iota and spanl

The function `iota` is particularly handy if you wish to graph a function at an equally spaced set of points.

```
iota(n)
```

will give you a vector whose components are `1`, `2`, `3`, ... , `n`.

```
iota(m,n) #or iota(m:n)
```

will give you a vector whose components are `m`, `m1+`, `m2+`, ... , `n`. Thus, for example, you can get a vector of all the points a tenth of a unit apart in the unit interval `[0,1]`, and the corresponding values of a function `f`, by writing

```
real x = 0.1 * iota ( 0 , 10 ) , y = f ( x )
```

Note that in Basis you need not specify the dimension of a variable that is initialized with a vector or array when it is declared. It will be automatically dimensioned properly (with all subscripts based at 1). Note that `real x = 0.:1.:.1` would have accomplished the same result, as would `real x = 0.:1.:11`.

The function `spanl` is used to obtain a vector of points which are *logarithmically* spaced between two given points, rather than linearly spaced as one would obtain with `iota`. To get the eleven logarithmically spaced points in the interval `[0,1]` use

```
spanl ( 0., 1., 11 )
```

The first two arguments are the endpoints of the interval, and the third is the total number of points desired.

7.3 Information about Arrays: length, shape

Arrays are ubiquitous in Basis. Subscripting can be bizarre and shapes can change. These two functions allow you to obtain information about the size and shapes of arrays; the `shape` function also allows you to arrange the elements of an array into a different shape. This can be especially useful inside a function, where you may have been sent a perfectly arbitrary array as a parameter and you need to determine information about it.

7.3.1 The Function length

To find the size of (total number of elements in) an array, takes its `length`:

```
real a ( 3, 8 , 7 ) , b ( 5 , 5 )
function howbig ( x )
remark length ( x )
endf
call howbig ( a )
call howbig ( b )
```

will print out 168 and then 25. (`remark` is a Basis macro which simply prints out the value of its argument at the terminal.) Do not confuse `length` with `strlen`, which counts the number of characters in a character string.

7.3.2 The Function shape

The "shape" of an array is defined to be a vector containing the span of its subscripts as its components. The span of a subscript is the total number of values which it can assume, which if its upper bound is `hi` and its lower bound is `lo`, is given by `hi - lo` 1+. The function `shape` can do two distinct things for you; the first one is that if you send it a single argument, it will return you the shape vector for that argument. For example, with the above `x`, the value of `shape(x)` would be `[4,6,9,4]`.

The function `shape` also can be used to take a given array and change it to an array with a different shape. In this case we need to send it the shape vector of the result, as a second argument (or send the components of the shape vector as additional scalar arguments). For example,

```
shape (iota(64) , [4 , 4 , 4])
#or shape(iota(64),4,4,4)
```

will return a 4 by 4 by 4 array whose components in column major order are the numbers $1, 2, 3, ..., 64$. Why might one want to change the shape of an array? Well, one application that comes to mind is that you might want to add two arrays together componentwise, and they are the same length, so it ought to be possible. Unfortunately Basis will not allow you to perform binary operations on objects of different shapes. So you need to coerce one of the objects to the same shape as the other. For instance, suppose `a` is a 5 by 5 array and `b` is a 25 element vector, and we wish to add `b` to `a` componentwise, and leave the result in `a`. We could do this as follows:

```
a = a + shape ( b , 5 , 5 )
```

The shape of b is not permanently changed by this operation.

The shape function can also replicate an array to fill up a larger shape:

```
shape([1,2], 2, 3) = [[1,2],[1,2], [1,2]]
```

and

```
shape(1., shape(a))
```

is an array of 1.'s shaped like `a`.

7.4 Summing Arrays: sum

The function `sum` allows you to add up the elements of an array without having to write a loop to do it. It takes one or two arguments; in the one-argument case,

```
sum ( x )
```

all the elements of array `x` will be added, and the scalar sum returned. if `x` is a scalar, `x` will be returned.

In the two-argument case, the second argument specifies a subscript of the first; the result will be an array containing the sums of the elements of the original over all values of that subscript. Thus the result array will be one dimension smaller than the original, but the same shape in the other dimensions. For instance, if `x` has shape `[12,8,90,10]`, then

```
sum ( x , 3 )
```

will sum `x` over the third subscript and produce a result having shape `[12,8,10]`. That is, `sum(x, 3, y)(i, j, l)` is the sum of `x(i, j, k, l)` over all `k`.

Likewise, if `y` were a two-dimensional array (i.e., a matrix), then

```
sum ( y , 2 )
```

would produce a vector whose components were the sums of the corresponding rows of matrix `y`.

If you like `sum`, you might also like its cousin `psum` (partial sum) which is good for integrating things.

7.5 Vector Conditionals with where

```
where ( cond , x , y )
```

The first argument `cond` must be of logical type and the second and third, `x` and `y`, must be of numerical type. The length of `cond` must be matched by that of `x` and `y`, although one of them can be a scalar and the other an array. The array returned consists of an array the same length as `cond` with components equal to the corresponding component of `x` for those elements of `cond` which are true, and the corresponding element of `y` where `cond` is false.

`where` also has a two-argument form which returns just those elements of `x` for which `cond` is true (a “compress”).

```
where ( a > b , a , b )
```

is equivalent to `max(a,b)` because `a > b` is an *array* of logicals of the same size and shape whose *components* are `true` or `false` according as the *corresponding components* of `a` are or are not greater than those of `b`. Thus `where` will now return an array whose components are the larger of the components of `a` and `b`. You could do the same thing with `max`; but the point here is that the logical condition could be a great deal more complex. In other words, `where` allows you to build much more general functions than `max` and `min`, although only on two arguments.

CHAPTER

EIGHT

Commands

8.1 The Basis Command Capability

Frequently when you are using Basis with a simulation code of some sort, the author will have written a number of commands which you will be using. In Basis, a command looks much like a command line in some operating system: the name of the command, followed by a list of arguments separated from one another somehow (usually by spaces or commas, but not always). A number of questions commonly arise with commands, in particular:

- What are the types of the arguments? (Specifically, some may be expressions to be computed, and others may be strings to be taken literally. Which are which?)
- What delimiters are allowed or required? (Clearly, if spaces are delimiters, then "`3 +4`" is two arguments, whereas if they are not, then it is one argument.)

Let's consider a little background first. Basis has a built in command capability that allows any function to be invoked by a command-line type of syntax. Consider, for example, a function defined as in a previous chapter:

```
FUNCTION w(namex)
INDIRECT y=namex
y(3) = 7.
ENDF
```

Using the regular Fortran-like Basis syntax, this function is invoked by the `call` statement:

```
call w("x")
```

Basis has a reserved word "`command`" which allows any function to be invoked by a command line syntax. In this example, `w` would be invoked by the statement

```
w command "x"
```

That is, we give the function name, the reserved word `command`, and then a list of the function's arguments. If there is more than one parameter on this list, the list may be delimited by either commas or spaces. (If this seems an arcane way to call a function, just remember that virtually all operating system commands have essentially this form: name of command followed by list of arguments.)

Using this command syntax, the arguments are all evaluated as expressions, and the default delimiters are spaces and/or commas. For example, in the command line

```
foo command "This is a string"  756 , , ( 912 + y )
```

the function `foo` is being called with four arguments:

- the character string `"This is a string"`
- the numerical value `756`
- a null argument (between the two commas)
- the expression `( 912 + y )`

The first two arguments are delimited by a space; the remaining ones are delimited by commas (spaces on either side of the commas do not count as delimiters when commas are present).

`NOTE:` Spaces outside parentheses act as argument separators; spaces inside do not.

It is important to emphasize that all of the arguments of `command` are taken to be expressions (in the above case, string-valued, numerical-valued, null-valued, and numerical-valued, respectively). Commas and spaces are taken to be delimiters (though not in combination—spaces around commas are ignored). Character string expressions must be quoted. If we had written

```
foo command This is a string  756 , , ( 912 + y )
```

then all of a sudden we would have seven arguments, and `This`, `is`, `a`, and `string` would be taken as identifiers to be evaluated.

The command capability allows the author to change these defaults. The author may specify different delimiters, either for the entire command or just between certain arguments; and can specify that some arguments be treated as if they were quoted strings, even if the quotes are not physically present. The details of this are covered in Chapter 10, "Deciphering Commands". This is done by suffixing an underscore "_" to "`command`" and then a sequence of letters specifying delimiters and argument types. Here is what we could do in the above case:

```
foo command_wSe This is a string , 756 , ,  ( 912 + y )
```

Immediately following the underscore is the lower case " `w`", which suppresses white space as a default delimiter. Thus only commas are valid delimiters in what follows. The upper case " `S`" specifies that the first argument (everything up to the first comma) is to be taken as a string, i.e., to be treated exactly as if it were quoted. The lower case " `e`" specifies that the second (and all remaining) arguments are to be expressions. Notice that since white space has been suppressed as a delimiter, parentheses are no longer necessary in the last expression.

What the author does to hide all of this from you is to define the command to be a macro which expands into the `foo command_wSe`. You can see what a command name really stands for by using the (you guessed it) `LIST` command.

In the chapter "Deciphering Commands" we go into more detail on this subject.

CHAPTER

NINE

Saving and Restoring Code and Data in Binary

9.1 The PFB Package

The PFB package can save and restore data, functions, and macros in binary form. The PFB package is not a required component of a Basis program; use '`list packages`' to see if it is present. (Note for when you start writing your own programs: PFB can be added to a program you make with Basis by adding the name pfb to the directory list input for `mmm`.)

Basically the process has three steps.

1. Create an output file:

   ```
   create myfile    # or whatever
   ```

2. Enter one or more `write` commands, which can take the following forms:

   ```
   write <namelist> # saves all the items named
   ```

   There are the following special forms of this command:

   ```
   write functions   # saves all user-defined Basis functions
   write macros      # saves all currently defined macros
   write variables   # saves all user-defined variables
   write all         # save functions, macros, and variables
   ```

3. When finished, close the file:`close`.

   The data is stored in a portable database format named PDB. The files can be moved to another computer and used there even if the new computer has a different data format.

9.2 Reading in Previously Saved Data

To read in all of the data you wrote do:

```
restore myfile            # reads all data from file myfile
```

To examine the data in the file without bringing it into your program permanently, you can use the `open` command:

```
real(8) x=3., y=4., z=5.
create myfile
write x,y,z
close
forget x,y,z  #x,y,z gone now
open myfile
x, y, z      #prints x,y,z in file
real(8) x=pfb.x #copy in just x
close
```

(You cannot assign to the variables in a file you have `opened`, you can only read the values).

See the manual page for PFB for fancier uses, such as comparing items from different files.

CHAPTER

TEN

Error Recovery and Diagnosis

10.1 Error Recovery

When an error occurs, it can be an error which Basis detects (such as trying to add a complex number and a character string) or one which is detected outside of Basis (such an floating point overflow). All errors detected by Basis result in a call to the routine `kaboom`. Whichever other errors a particular version of Basis can trap are trapped to a routine called `yuck` which in turn calls `kaboom`. This is why you may see a message '`yuck: floating point error`' followed by messages about recovering to the prompt. Very rare but serious errors may cause an immediate program exit via `baderr`, and some system errors cannot be trapped, and exit without allowing Basis to regain control.

Assuming you reach `kaboom`, it either returns you to the prompt or causes the program to terminate. By default it returns you to the prompt. A routine `errortrp` is provided which can change this behaviour:

- `errortrp("on")` causes `kaboom` to recover to the prompt.
- `errortrp("off")` causes `kaboom` to terminate the program.

When an error occurs a trace file is written containing diagnostic information. To help you with a problem, the Basis staff needs to know what messages exactly were printed on the terminal when the error occurred, and what information is in the trace file.

To increase the information you get when an error occurs, we recommend setting

```
debug = yes
```

at the beginning of your session. If the trace files just annoy you no end, you can set `bastrace="none"` to eliminate them but this will make it hard for us to help you.

10.2 Syntactic and Semantic Errors

There are two kinds of errors that Basis can find.

- *Syntax* errors occur during the parsing of input code, and are caused by grammatically incorrect statements. Typical errors might be an illegal character in the input, a missing operator, two operators in a row, two statements on the same line with no intervening semicolon, unbalanced parentheses, a misplaced reserved word, etc.
- *Semantic* errors occur during the execution of the code, after it has been parsed as grammatically correct. These have to do not with how statements are constructed, but with what they mean. Such things as incorrect variable types or sizes, nonexistent variables, subscripts out of range, and the like, are semantic errors.

Basis is a single-pass parser, that is, it looks at its input only once. It also is a one-look ahead parser, meaning that at the most it is never looking more than one symbol ahead of the current context. By the time a syntax error has been detected, it is likely that a lot of the context information to the left of the error has already been lost. The diagnostic information that Basis gives attempts to be as useful as possible, but because of the very limited context information available, it is far from perfect.

Semantic errors are often possible to diagnose more precisely. We have attempted to make the semantic error information supplied as useful as possible. Sometimes some of the information is only useful to someone familiar with the internals of Basis; but we hope that in most cases it will help you find your error.

10.2.1 Syntax Errors

Here is an example of a statement containing a syntax error:

```
sum ( where ( a > v , ones ( length ( a ) ) , 0)
```

Let's take a look at what Basis prints out as a result of this error:

```
sum ( where ( a > v , ones ( length ( a ) ) , 0 )
                                                 ^ Syntax error.
Attempting to parse after following context:
<lhs> ( <argitem>
which may not be followed by "cr" in this context.
Count of parentheses unbalanced: left = right +    1.
Expected one of the following (?):
  ) ,
Returned to user input level.
```

When the parser echoes the line being parsed, with "`Syntax error`" underneath the line, the caret points to where the error was detected, not necessarily to where it occurred. In this case, the caret points past the end of the line, a clue that something is missing. The information about the parsing context is useful only to a Basis expert, but the statement that it can not be followed by

"cr" (carriage return) is useful. That seems to say that the line is too short and reinforces our suspicion that something is missing. The next line points out that so far in the line there have been more left parentheses than right, and the next two lines confirm that maybe the parser expected a right parenthesis or a comma. The expression was missing a right parenthesis.

The list of expected symbols (as opposed to the one which actually occurred) is not 100% accurate. It may not contain all possible symbols which could occur in the given context; or worse yet, it could be such a long list as to be virtually unusable. In the above case it did contain the missing symbol, and it was not needlessly long. Below is a case where the list supplied by the parser is too extensive to be much help:

```
function f(x)
if ( x  > 0 ) then return 0
return 1
endf
```

The diagnostic produced by this error is:

```
     endf
     ^ Syntax error.
Attempting to parse after following context:
function <funcdes> <eos> <stlist> if <ifexp> then <stlist>
which may not be followed by "endf" in this context.
Expected one of the following (?):
  ( + - : << >> ? Groupname [ ^ ‘ break call chameleon character
complex complex-constant cr do double double-complex-constant
double-constant else elseif endif for forget function
hex-constant if indirect integer integer-constant list logical
name next octal-constant range read real real-constant return
string while whitespace \{ Returned to user input level.
```

What has happened here is a relatively common error—the programmer has not completed an IF statement. An ENDIF or ELSE clause has been omitted. Deeply buried in the list of “expected” symbols you will find these two reserved words, and also ELSEIF. It is possible to imagine a meaningful continuation of the program starting with any of the other symbols in the list, but the length of the list quite effectively hides the real clues in its depth. Unfortunately, a one-pass, no-backtracking parser with a one token lookahead can not apprehend the entire surrounding context as a human can; it only knows what symbols might, in some circumstances, lead to a correct statement if placed in the current position.

This example also hints at another problem with syntax errors: they may be discovered long after the actual error occurred. In this case, if an

ENDIF was intended prior to the return 1 statement, the error was not detected until the ENDF was seen, after that statement had been consumed. There could equally well have been a hundred statements parsed before the ENDF caused the parser to detect the error. Thus our advice is that

if you have trouble tracking down a syntax error, don't confine your search to the immediate neighborhood where it was detected. It could have been many lines previous.

10.2.2 Semantic Errors

Many times the Basis diagnostics for semantic errors make it very easy to discover what was wrong. For example, the statement

```
b = c + a
```

produces the following diagnostic (when `debug` is `yes`):

```
parasgn2: Shape mismatch between source and target in
assignment or append.
Right side (source) true dimension=  2 true shape=    10   10
Left  side (target) true dimension=  0 true shape=
parasgn: error in assignment to variable named 'b'.
Writing traceback info to file trace24589x
Returned to user input level.
```

Clearly the problem is that the right side is a 10 by 10 array, and the left side `b` is a scalar, so this is an illegal assignment.

Sometimes the traceback information can be useful; if you examine the traceback file (in this case `trace24589x`), you will find that it contains

```
Here is the information I have on where you were:
The error occurred in the assignment or append statement:
b = expression
The following lines contain clues(not facts) about the r. h. s.
c+a
Parser's action number =   3(ASSIGN  ), program counter =    26.
```

Frequently a semantic error will be detected inside a function, or perhaps nested inside several function calls. The error printout may concern a variable or parameter local to the function where execution is taking place, and the name of the variable seems totally off the wall. For instance, consider the declaration and function call:

```
integer z ( 2 , 4 , 5 )
call f ( z )
```

This function call produced the following error diagnostic:

```
parfetch: trouble with object named 'barf'.
expression being subscripted has     4 subscripts but variable
only has 3 dimensions.
Writing traceback info to file trace24589x
Returned to user input level.
```

Where did the object named "`barf`" come from? Clearly it has three dimensions but four subscripts, but we can scarcely correct the error until we know where it was. In this case, the traceback file proves invaluable. It contains (in part):

```
Here is the information I have on where you were:
   A call to f containing
   A call to brf containing
   A call to arf containing
   the problem.
Error occurred in non-assignment statement.
The following lines contain clues to the error.
Some or all may be irrelevant to your problem.
1
barf
crf
crf
arf(crf)
arf(crf)
brf(x)
z
f(z)
Parser's action number = 374(OUTPUT), program counter = 50.
Group: Locals_arf  Num Vars: 1
barf(2,4,5)
Number of dimensions is  3, lengths =      2     4     5
# # # some information skipped here
Group: Locals_brf  Num Vars: 1
crf(2,4,5)
Number of dimensions is  3, lengths =      2     4     5
# # # some information skipped here
Group: Locals_f  Num Vars: 1
x(2,4,5)
Number of dimensions is  3, lengths =      2     4     5
# # # some information skipped here
```

The function `f`, which we called, has called function `brf`, which called `arf`, where the error actually occurred. So the variable `barf`, which caused the trouble, is local to the innermost function `arf`, as we find out farther down the traceback. Finally, our variable `z`, and `f`'s variable

`x`, and `brf`'s variable `crf`, and finally `arf`'s variable `barf`, all have the same dimensions. It must be that these are the names of these functions' formal parameters; `z` has been passed down as a parameter all the way to a routine which expected a variable with four dimensions. Either we declared `z` wrong, or misunderstood what number of dimensions it was supposed to have, or else there is an error in `arf` which needs to be corrected.

Error diagnosis is (usually) a fairly straightforward problem, and we hope that these examples have helped illustrate how to diagnose and correct bugs. Our final advice is:

- Set `debug` to `yes` to get the maximum information.
- If the terminal output is not adequate to locate the bug, use the traceback file. (You can see its contents by typing

  ```
  !more tracefilename # name is given in diagnostic
  ```

  at the basis prompt.)
- Please be patient and actually read the error messages and trace files. We find many users do not do this. Errors, especially when you are busy, can generate strong emotions. If we knew how to generate one-line messages that exactly described every error, we would. We need to put out a lot of information to help people find difficult bugs; this means that often a lot of information is put out about a simple bug. We have tried to recognize this problem and put the more elaborate information in the trace file, so that it isn't read except in more difficult cases.

CHAPTER

ELEVEN

Deciphering Commands

This chapter continues the discussion of commands. We do not recommend reading this chapter on a first reading of this manual. Or a second. Come here when you have a real problem with a command.

Commands defined by the author of a code are defined as macros. You need not know how to write a macro yourself in order to understand one somebody else wrote. A macro definition associates a name with a body of text, and this text is substituted for the macro name whenever it is encountered in the input stream. Macros may have arguments, in which case the arguments are expanded where they occur in the macro text. Thus a macro invocation can look just like a function call.

There are a few exceptions to macro expansion:

1. Text inside quoted strings is never expanded.
2. Macro expansion in an expression can be suppressed by enclosing the expression in braces { and }. (The braces are otherwise ignored by Basis.)
3. Macro expansion in a command argument can be suppressed by expressing its type with an upper case letter (in the preceding example, "`S`" as opposed to "`e`"). More about this later.

Enter this and see what happens:

```
list pi
pi
{pi}
```

the `LIST` command causes the macro definition of `pi` to be displayed. The second line will cause a funny display something like

```
3.14159265358979323 =    3.14159D+00
```

Left of the `=` sign is the text that was actually substituted for `pi`. Right of the `=` sign is the value of this, to the number of digits specified by the built-in variable `fuzz`. The third line causes an

error because if we suppress the macro expansion of `pi` by enclosing it in braces, it then becomes an unknown symbol.

What does all this have to do with deciphering somebody's command? Well, suppose you are running somebody's simulation code with the Basis interface, and there is a certain command you want to use, and you are unsure what the arguments are supposed to be or how they are supposed to be delimited. Typically this individual will have defined this command as a macro, so the first thing you need to do is to track down the text of this macro. This is not hard to do; simply type in

```
list commandname
```

Basis has a command `timer` which is used as `timer on` and `timer off`. The text for `timer` is:

```
partime command_s
```

This means that `timer` calls a function named "`partime`." The "`command_s`" specifies (via the "`_s`") that it accepts at least one argument and that the argument will be an unquoted string with macro expansion enabled. "`s`" and "`S`" both express that an unquoted string argument is expected; upper case causes macro expansion in the argument to be suppressed. We don't know from the definition how many arguments the macro (or function) expects; but they will all be unquoted strings, if there are more than one—this is governed by the last letter in the specification (and here, the only letter, "`s`"). There are no delimiter specifications in this command, so white space and commas will be accepted. By the way, string arguments can be quoted, if you wish; they just don't have to be—*unless* they are to contain symbols that would be recognized as delimiters.

Argument specifiers can be `s` or `S` for strings (with and without macro expansion), and `e` or `E` for expressions (again, with and without macro expansion, but `E` is hardly ever used). You can specify a type for every argument by having a string of these characters, one per argument; but if, as is often the case, all the arguments from some point on are the same type, then Basis will keep using the last character in the string of specifiers. Thus `command_se` is the same as `command_seeee`... . Parentheses are used to specify repetition of more than one type, e. g., `command_e(Se)` is the same as `command_eSeSeSe`... .

Delimiter specifiers may be included in the specification string. If they are at the very beginning of the string then they determine the default delimiters for all arguments. If they occur between argument specifiers, they express what delimiter(s) would be valid just between those two arguments. Delimiter specifiers are `w/W` (suppress/enable white space), `c/C` (suppress/enable comma) `a/A` (suppress/enable at sign), and `q/Q` (suppress/enable equal sign). As has been previously mentioned, if no delimiters are specified then the default is "`WCaq`", i.e., white space and comma enabled, at sign and equal sign disabled.

Let us look at a few more examples from among the Basis predefined macros. Here is the expansion for `tek` in the `ezn` graphics package:

```
ezcdodev command_S(ScQS) tek $1
```

This calls a function named `ezcdodev`. The first argument is a string with no macro expansion (since this is the string "`tek`" itself, it is clear that we do not want to expand it in its own expansion). Subsequent arguments (if any) occur as pairs of strings with no macro expansion; the two arguments in a pair can be separated by equal signs or white space, but not commas (the "`cQ`" specification disables commas, enables equals, and does not change white space, which is enabled by default). Pairs are separated from other pairs by the default, white space or commas, because the "`cQ`" specification occurs only between the elements of a pair. The "`$1`" notation stands for the first argument of the macro call. (A macro may be defined with arguments, just like a function, in which case, when an invocation of the macro is expanded, `$1` will be replaced by the text of the first actual argument.)

Here is the macro text for `cgm`:

```
ezcdodev command_SSc(SwcS) cgm $1
```

this calls the same function, but the arguments and delimiters are specified differently. All arguments are strings with no macro expansion. The first two are separated by white space or comma (the default), and the second is separated from the third by white space only ("`c`" suppresses comma). Subsequent arguments, it would appear, occur in pairs with white space or commas between the pairs, but the puzzling "`wc`" seems to say that the two strings of a pair have no delimiters between them at all! On first glance this seems to make no sense; but in fact, the effect of this is to concatenate the entire rest of the line into a single string and never find a fourth argument. Thus, in fact, this specification is really the same as

```
ezcdodev command_SScSwc  cgm $1
```

since no fourth argument will ever be collected; there is no delimiter possible to set it off from the third.

The following is the expansion of `plotm`:

```
ezcplotm command_(eWCQ)
```

All the arguments of this command will be expressions with macro expansion enabled, and the delimiters will be white space, commas, and equal symbols.

Finally, here is `resume`:

```
osresume command_es $1 $2
```

This calls the function `osresume` with the macro's first two arguments (`$1` and `$2`), the first of which is an expression, and the second of which is a string, with macro expansion enabled in both. The delimiters are default.

Part II

Basis Language Reference

CHAPTER

TWELVE

Basis Input

Basis input can come from the terminal, from a file, or via recursive calls from within compiled code that is being executed. Statements are executed one at a time, immediately. However, statements which are part of a larger construct (such as a loop or IF test) are not executed until the entire construct is complete. When interactively entering such constructs, the prompt will change to give you a visual clue to the depth of the structure in which you are presently.

When an error occurs in a series of statements, you can assume that the statements before the one that caused the error have been executed; but if you make a mistake when entering a more complicated structure, you will need to begin again from the beginning. For example, if you are defining a function, and enter a statement that has improper syntax, the preceding part of the function is lost.

For this reason, it is usual to place complicated Basis input in a file and use the READ command to process it.

Basis-reserved words (like READ) are written in upper case throughout this manual for purposes of emphasis but they are also recognized by Basis if they are entered entirely in lower case.

CHAPTER

THIRTEEN

Basis Tokens

13.1 What Is A Token?

The tokens or terminal symbols of a language are the basic building blocks of that language. Tokens are the lexical entities from which statements in that language are constructed. They are analogous to the words in a spoken language. It will help in the discussion of the Basis Language to have an idea of what its tokens are before studying the language syntax. Tokens can be divided into the following categories:

- Alphanumeric These include identifiers and constants.
- Reserved words Identifiers that have a special meaning and may not be used in another way, such as the word IF.
- Non-alphanumeric These include punctuation, separator symbols, operators, and the like.

Comments and line-continuation symbols are not language tokens; they are delimited by symbols that have no significance in the Basis Language. The Basis-reserved words, built-in functions, non-alphanumeric, and alphanumeric tokens are described in later sections.

13.2 Special Characters

Some of the non-alphanumeric tokens have special interpretations in Basis:

- Blanks and spaces are significant in Basis. They act as token separators. The number of blanks (spaces) is irrelevant, however, as long as there is at least one. Thus, for instance, ELSEIF is one token; ELSE IF is two tokens.
- Semicolon and carriage return (the end-of-line character) may be used interchangeably by the user as statement separators.

- If \ (backslash) occurs as the last character on a line, the next end-of-line is ignored, and so allows continuation from one line to the next. This is the only way to continue a quoted string. A line is also continued if the last token on a line is a left parenthesis, a comma, or any logical or arithmetic operator such as `+, -, *, !, &,` etc.

- #acts as a comment delimiter, and causes the rest of the current line to be ignored except for the end-of-line. None of the special characters has any special meaning inside quoted strings or in comments.

13.3 Alphanumeric and ConstantTokens

13.3.1 Identifiers or Names

The names of variables must begin with a lowercase letter or a dollar sign. Subsequently, names can contain letters of either case, underscores, or digits. Names of variables must be 128 or fewer characters. Case is significant, so `joe` and `joE` are distinct variables.

A name can also be specified by enclosing it in single right quotation marks (apostrophes). In that case the name can include any characters except apostrophes, carriage-returns, or line-feeds. The enclosing apostrophes do not become part of the name; they simply allow names which do not obey the above rules to pass the interpreter.

Different variables can have the same name, if they appear in different packages in the search stack. (Please see “The Search Stack” on page 25 for a discussion of the search stack.) An identifier is taken to be the first one encountered in a package that has that variable name. To access a variable that is not in the top package or to distinguish between variables with the same name in different packages in the stack, add the package name as a prefix, and separate the package name and variable name with a period. For example:

```
pkg.name2
local.name3
global.name4
```

The variables in packages are organized into groups. A group name must begin with a capital letter, and again, can be prefixed with a package name followed by a period to identify it uniquely. *Global* and *local* are legitimate package names for user defined global and local variables.

Variables from packages attached to Basis are organized into groups. Group names can be up to 128 characters and can be abbreviated to any unique prefix. Any variables the user declares become members of a special group called *User*.

The identifiers `$`, `$a`, `$b`, `...`, `$z` are pre-declared. They have the chameleon property, which is discussed in “Declaring and Initializing Variables” on page 14. The variable `$` always holds the value of the last expression displayed.

13.3.2 Constants

An integer constant is a string of one or more digits, as in Fortran. A real constant in the form xx.xE+x must contain at least one digit, and either a decimal point or an exponent, or both. The exponent is expressed as e or E followed by an optional sign and at least one digit. Imaginary constants are either an integer or real constant followed by i or I. Spaces are *not* allowed between the number and the imaginary notation. Thus, 3I, 3.0i, 0.3E+1I, all represent the same imaginary constant. Double-precision constants are the same as real constants except that the letters d or D are used to denote the exponent.

String constants are delimited by double quotes (`"`). They must contain at least one character and can be composed of any printable ASCII characters; to include a double quote in a string constant, double it.

Two special forms of integer constants are also available: octal and hexadecimal constants. An octal constant is an octal number followed by b or B. A hexadecimal constant is a hexadecimal number, beginning with one of the digits 0-9, followed by an x or X.

Basis also contains the variables listed in “List of Parser Variables”, on page 42. These variables, such as “`pi`”, are available to the user for use in statements.

CHAPTER

FOURTEEN

Declaring and Initializing Variables

The name of a user declared run-time variable must begin with a lower-case letter.

Users can declare run-time variables to be of type INTEGER, INTEGER(4), INTEGER(8), REAL, REAL(4), REAL(8), DOUBLE, LOGICAL, COMPLEX, COMPLEX(4), COMPLEX(8), CHARACTER, CHARACTER*(n), RANGE, INDIRECT, or CHAMELEON. The types CHAMELEON and INDIRECT are discussed in the following sections. Types REAL8 and COMPLEX8 (no parentheses) are also available, with the same meaning as REAL(8), COMPLEX(8), respectively.

Variables can be initialized in the declarations statement, as shown in the scalar declarations below:

```
INTEGER     x, y, z
INTEGER(4)  i4
INTEGER(8)  i8
REAL        i, j, k = 2.0
REAL(4)     x4
REAL(8)     x8
DOUBLE      d  = 2.d0
COMPLEX     c  = 2.0 + 3.0i
COMPLEX(4)  c4
COMPLEX(8)  c8
LOGICAL     l1 = true, l2 = false
CHARACTER*3 ch = "abc"
```

The variables x, y, and z are declared as integers of default size. I4 is an integer at least 32 bits (4 bytes) in length, and i8 is at least 64 bits (8 bytes) long. Variables i, j, and k are of type default real; k is initialized to 2.0. The variables x4 and c4 have at least 32 bits (4 bytes) precision independent of platform, and x8, c8 are at least 64 bits (8 bytes) in size.

Basis' use of *kind selectors* for integer, real, and complex data types is very similar to their use in Fortran 90. The discussion of precision above presumes that the underlying hardware is based on twos-complement integers and IEEE 754-standard floating point representations, in which case `real(4)` corresponds to IEEE single precision and `real(8)` to double. To restate this in Fortran 90 terms, a Basis `real(4)` kind should be the same as that resulting from `kind = selected_real_kind(6,38)`, and `real(8)` should match `kind = selected_real_kind(15,308)`.

Each individual variable to be initialized must be followed by an equal sign and value. To initialize `i`, `j`, and `k` to 1, 2, and 1, respectively, enter the following:

```
INTEGER i = 1, j = 2, k = 1
```

Variables which are not explicitly initialized are set to 0, or to blanks if they are of character type.

The variable called `autovar` controls whether or not declarations are required for all variables. See “autovar” on page 179.

Declare array variables of up to seven dimensions as follows:

```
REAL x(10), y(3,5), z(-3:5, 7:10)
```

The lowest value of the subscript range defaults to 1 unless a different value is specified before a colon, as in `z` above. Thus, `x` is subscripted 1... 10, `y` from 1... 3 and 1... 5, and `z` from -3... 5 and 7... 10. An individual array can be initialized by a vector of values that follows its type declaration:

```
INTEGER i(10) = [0,0,0,0,0,1,1,1,1,1], j(5) = [1,2,3,4,5]
```

Vectors cannot be larger than the variables they initialize (except see the next paragraph), but they can be smaller, in which case only the first specified number of positions in the array will be filled. The initialization in a declaration follows the rules for assignment statements.

If an initial value is given, but no dimension is given on the variable being declared, the variable is created with the dimensions of the initial value. Thus the previous example could also be done this way:

```
INTEGER i = [0,0,0,0,0,1,1,1,1,1], j = [1,2,3,4,5]
```

Basis allows initialization expressions of arbitrary complexity, as long as all operands in them have values at run-time, since in fact such statements are ordinary assignment statements. References to functions are allowed as well. For example,

```
REAL a = sqrt(2) * ones(10,10)
```

defines a diagonal matrix `a` with the square root of two on its diagonal. For more details on expressions, see the next section in this manual, “Basis Expressions”.

The dimension specifications of declared variables are also allowed to be expressions of arbitrary complexity, as long as they are capable of evaluation at the time the declaration is executed. For example,

```
INTEGER i = 5, a(i,0:3*i-2) = 4, b(a(1,0))
```

declares and initializes `i` to 5, and then declares a to be an array subscripted 1...5 and 0...13, and then declares `b` to be subscripted 1...4.

As remarked previously, reserved words cannot be used as user identifiers. Previously declared variables or functions can be redeclared at any time, however. If the parser variable `debug` has been set to yes, Basis prints a warning message when a variable or function is redefined.

14.1 GLOBAL declarations

When a variable is declared it normally has global scope, that is, it will be known inside user-defined functions without any further declaration. If, however, a declaration occurs inside the definition of a user-defined function, the variable becomes local to that function invocation and will not be visible outside of it, and will vanish when the function returns. The user may override this by prefixing the keyword GLOBAL to any declaration inside a function, thus creating a variable which will be identical in scope to one declared outside of any function. For example,

```
FUNCTION phi(z)
  global REAL x = z/2.
ENDF
```

will create a variable `x` when function `phi` is called. Any existing global variable named `x` will be destroyed.

14.2 Package declarations

In addition to declaring global and local variables, the user may also declare a new variable to reside in an existing Basis package. When such a declaration is made, the variable is put in the last group of that package. Such a variable would be declared by a statement of the following format:

```
pkg type varname
```

where `pkg` is the name of the package in which to create the variable, `type` is the type of the variable (such as real or integer), and `varname` is the name or the variable.

EXAMPLE:

```
par REAL x = 3.1
```

The above example will create variable `x` in package `par`. The user can determine which packages exist in a given Basis code by typing,

```
LIST packages
```

14.3 Chameleon Variables

The variables `$` and `$a, $b, $c, ..., $z` exist when Basis starts. The variable `$` automatically assumes the value of the last expression evaluated in a display statement; the others must be explicitly assigned (but see the variable `autohist`, page 179.) When assigned a value, these

variables assume all of the attributes (e.g, type, size) of the value assigned to them. Hence, they are called chameleon variables. The user may declare other variables to have this chameleon property by using the type CHAMELEON, e.g.,

```
CHAMELEON abc = 3.45
```

causes `abc` to become a real whose value is 3.45. If a chameleon variable is currently an array then a subscripted assignment to the variable behaves like a normal assignment statement. Thus,

```
CHAMELEON abc = [1,2,3,4]
abc(3) = 5.6
```

results in `abc` being equal to `[1,2,5,4]` because the first assignment statement makes `abc` an integer array of length 4, and the second assignment statement has a subscript on `abc`, so its chameleon property is not invoked and the 5.6 is coerced to integer before being stored.

Except for `$` and `$a, ..., $z`, all variables that are assigned a value at execution time must exist, i.e., must have been declared. (The control variable `autovar` can be set to `yes` to change this). Formal parameters (the variables in the argument list) in user functions may not be declared.

14.4 Computed Names

It is possible to compute a name to be used in a declaration statement. This is done by surrounding a character expression with grave accent marks, as in this example which creates a variable named `x1` and initializes it to 3.0:

```
real  ‘ "x"//"1" ‘ = 3.0
```

14.5 Range Variables

A RANGE type is the same entity as the range used to subscript a variable. It consists of a low index, high index, and possibly an increment (negative increments are allowed), all separated by colons. An integer is also accepted as a range in which the low and high index are the same value.

EXAMPLE:

```
RANGE x = 3:5, y = 1:5:2, z = 5:2:-1, zz = 4
```

RANGE variables would eventually be used as subscripting information for an array. However, these variables can be passed in as arguments to a function and used within that function.Subscripting using a RANGE variable is identical to direct subscripting. Thus RANGE variables can have defaulted fields for their low index, high index, or increment (i.e. RANGE x = ::3).

The defaulted fields will take on the appropriate values for each array it subscripts. Some simple operations can also be performed on RANGE variables. You can add, subtract, or compare (i.e. ==, ¡>) two RANGE variables.

Three sets of examples and descriptions follow to illustrate

1. adding and subtracting RANGES, and the rules governing these operations
2. subscripting with RANGES and using DEFAULT fields
3. passing RANGES as arguments to functions

EXAMPLE OF RANGES WITH DEFAULT FIELDS:

```
RANGE a=2:10, b=: :3
integer z(a), y(6,7)
z(b)
y(4,b)
```

The above example will declare an integer vector `z` which is indexed from `z(2)` to `z(10)` and a 2D integer array dimensioned 6 x 7. The line "`z(b)`" will cause the values of `z(2)`, `z(5)`, and `z(8)` to be printed (just as if you entered `z(::3)`). The line `y(4,b)` will cause the values of `y(4,1)`, `y(4,4)`, and `y(4,7)` to be printed.

It should be noted that if you print the value of a RANGE variable which has a default low or high index, then any defaulted indices will be printed as a large negative number. Defaulted fields in a RANGE variable do not take on the "correct" value until it is used as an array subscript.

EXAMPLE OF ADDING AND SUBTRACTING RANGE VARIABLES:

The precise rules for addition and subtraction of ranges follow the examples.

```
range a=3:4, b=2:7:2, c=10:6:-1
a+4     ## results in 7:8,    remember 4 is the same as 4:4
c-4     ## results in 6:2:-1
b+b     ## results in 4:14:2
a+b     ## results in 5:11:2
b+c     #### illegal operation
```

When adding or subtracting ranges, the low indices of the operands are added or subtracted to produce the new low index and similarly the high indices are added or subtracted to produce the new high index. However, the resulting increment field is calculated in a different manner.

If the increments of both operands are 1, then the resulting increment is 1. If one operand has an increment of 1 and the other operand has an increment not equal to 1, then the resulting increment is set to the non-one value. If both operands have an increment other than 1, then these increment fields must both be the same value or else the operation is illegal. The resulting increment field is the same value as increments of both operands.

WARNING: Before adding or subtracting RANGES, you should always first call Basis function RNGSETDF to set any defaulted fields in the RANGE variable to the correct values. Adding or subtracting ranges with defaulted values which have not been reset by RNGSETDF will produce unexpected results.

EXAMPLE OF RANGE VARIABLES PASSED TO FUNCTIONS:

```
function density(x,y);  return mass(x,y)/volume(x,y); endf
function diffa(x)
  x=rngsetdf(x,2:10)   ## replace any default values of range x
  return a(x) - a(x-1)
endf
density(2:4, 1:10:2)
integer a(10) = iota(10)
diffa(3:7)
diffa(3: )           ## default value of high index in 3: is 10.
```

The calls to `density(2:4, 1:10:2)`, `diffa(3:7)`, and `diffa(3: )` will only calculate those values which are given in the ranged subscripts. In addition, the function `diffa` shows an example of range subtraction. This function makes a call to `rngsetdf` (a Basis built-in function) to replace any default values before doing the RANGE subtraction. Thus in the case when argument x is 3:, then x is reset to 3:10 before doing the subtraction. The function then returns the values a(3:10)-a(2:9).

14.6 The Colon Notation For Vectors

The notation `a:b:c` can be used with one or more real arguments to create linearly spaced arrays.

a:b:c with c real, a or b real

creates a vector containing values spaced at intervals spaced `c` apart. If `a`>`b`, the resulting vector will contain descending values. The vector created will be at least 2 long, and the first element will be `a` and the last will be `b`EXACTLY.

a:b:ic with ic an integer, a or b real

creates a vector of length `ic` of evenly spaced values from `a` to `b`. If `a`>`b` the resulting vector will contain descending values.

a:b with a or b real

defaults `ic` to the value contained in the control variable `ncolon`, whose default value is 100.

It is an error for `a` or `b` to be omitted if the other is real. It is an error for `c` or `ic` to be $<= 0$.

Note that the colon operator has a lower precedence than arithmetic operators, so to use a term `a:b:c` in an expression it will usually be necessary to enclose it in parentheses.

14.7 Indirect Variables

A variable declared to be type INDIRECT is actually an indirect reference to another variable. An INDIRECT declaration must include an initial value assignment setting the variable to the name of another variable, possibly including a package prefix, such as `"x"` or `"par.x"`. Any reference to an indirect variable after its declaration is equivalent to a reference to the variable named in the initial assignment. This assignment can only be changed with another INDIRECT declaration.

The variable which is being indirectly accessed may in turn be an indirect reference. INDIRECT can be used to write user functions which modify variables in their argument list; normal Basis functions pass arguments by value and such modifications do not

```
REAL x(100)
FUNCTION w(namex)
  INDIRECT y=namex
  y(3) = 7.
ENDF
call w("x")
```

will result in `x(3)` being set to 7. By contrast,

```
REAL x(100)
FUNCTION w(y)
  y(3) = 7.   #THIS IS USELESS
ENDF
call w(x)
```

does NOT modify `x`; rather, a copy of `x` has been modified, and then discarded when `w` returned.

CHAPTER

FIFTEEN

Expressions

15.1 Introduction

Expressions consist of operands, operators, and delimiters in a string specified by the grammar. Conceptually, we can consider operands as items that have value (e.g., constants, references to user variables that have a value, and invocations of functions that return values when executed). Operators are syntactic tokens that are usually described in terms of their semantic meanings, (i.e., what they are supposed to do at execution time). Unary operators produce a value from one operand, binary operators from two operands, and ternary operators from three operands. Finally, delimiters separate items (e.g., a comma-delimited list) and to change the semantic meaning of what they enclose (e.g., parentheses that change the precedence of enclosed operators).

15.2 Operands

String constants can be assigned to a variable, concatenated, built into arrays, passed to functions as arguments, etc. Everything that follows in this section addresses numerical and logical computations.

There are two types of expressions in Basis: expressions with numerical values denoted here by <exp>, and more complicated expressions, which, because they are allowed to have either numerical or logical values, are denoted by <lexp>.

The operands in <exp>s can be any of the following:

1. Integer, real, double, or imaginary constants.
2. Scalar variables of type integer, real, double, or complex.
3. References to arrays of type integer, real, double, or complex.
4. References to functions that return scalar or array values of type integer, real, double, logical, string or complex. There are three kinds of functions:

(a) Built-in functions are special functions that have been built into Basis. These are discussed in "Built-in Functions" on page 101.

(b) Compiled functions are Fortran functions that have been entered into a package database so that they can be invoked through the interpreter.

(c) User-defined functions are functions in the Basis Language defined by user commands, as explained later.

A reference to a scalar variable consists simply of its name if it is a user-defined variable, or if it refers to the top-most variable in the package stack by that name. Otherwise, it is referenced by a name of the form `pkg.name` where `pkg` is the name of the package in which it is defined.

A reference to a function consists of the name of the function followed by a list of expressions for its actual arguments in parentheses, as in Fortran or Pascal. Built-in and user functions are referenced in exactly the same way. If the function has 0 as an acceptable number of arguments, parenthesis are optional.

The operands in <`lexp`>s can be any of the operands allowed for <`exp`>s. In addition, <`lexp`>s can have operands of logical type, including the logical constants TRUE and FALSE. In some cases, logical quantities can also be organized and referenced as arrays. Array references and values of all types are discussed later in this chapter after a thorough consideration of scalar expressions.

15.3 Operators

The unary arithmetic operations are + and -. These two symbols also denote the binary operators "add" and "subtract". They have the lowest precedence of all operators. This means that in expressions containing other operators, add and subtract are evaluated last, as long as parentheses do not change the order of precedence. In expressions containing more than one of these operators, they associate to the left, which means that an expression such as

```
a + b - c + d
```

is evaluated as if it had been written

```
((a + b) - c) + d.
```

The binary operators "multiply" (`*`) and "divide" (`/`) have the next highest precedence, and are also left-associative. Thus, for example, in

```
a*b + c*d,
```

both the products `a*b` and `c*d` are computed, and then the addition is performed. In the expression

```
b/2*a
```

b will be divided by two, and then the result multiplied by a. To divide b by two times a, the expression must be written

```
b/(2*a) or b/2/a
```

The (scalar) arithmetic operator of highest precedence is the "exponentiate" (**) operator:

```
a**3
```

means a^3. Unlike the other arithmetic operators, ** associates to the right, so that

```
a**b**c
```

is evaluated as if it had been written

```
a**(b**c)
```

Operands of real, double, integer, and complex types can be intermingled at will in arithmetic expressions. In expressions containing a complex operand, the result is forced to complex type; if only integers and reals are present, the result is real. In expressions containing only integers, the result is always integer. In the case of division with a non-zero remainder, the quotient is taken to be the integer part of the result. For instance, 17/3 has the value 5.

The "matrix multiply" (*!) operator has its own peculiar size rules. The dot product operator (!), applies to objects of equal size. The dot product operator is included in the discussion of array operands later in this chapter. Thus, for the time being, we have considered all of the scalar arithmetic operators. We now discuss <lexp>s, those expressions which may produce logical values.

Operands for <lexp>s, those expressions that may produce logical values, can be built in three ways: from the logical constants TRUE and FALSE; from relational (i.e., comparison) operators between arithmetic values; and by combining previously computed logical values with the use of the logical operators. The binary relational operators are:

Operator	**Meaning**
= or == or .eq.	"equal"
<> or ~= or .ne.	"not equal"
< or .lt.	"less than"
<= or .le.	"less than or equal"
> or .gt.	"greater than"
>= or .ge.	"greater than or equal"

The equal and not equal operators can appear between operands of arbitrary type. If the types do not match, then coercion takes place in the order *integer* → *real* → *double* → *complex*.

The other four relationals are not meaningful for complex operands, so they can be used only with real, double or integer operands. Only the equals and not-equals operators can be used between character strings.

If the operands are not scalar a relational operator produces a logical array of the same shape as the operands;

```
(iota(5)=iota(5))
```

creates a logical array of length 5, all of whose elements = TRUE.

WARNING: The parentheses are essential here.

The relational operators are not associative, so two or more cannot be used in combinations like `a < b <= c`. Many languages allow such constructs syntactically, but they are almost always erroneous semantically. If `a`, `b`, and `c` are numeric, `a < b` is logical, and it is not legal to compare the logical `a < b` with the numeric `c`.

Listed in order of precedence, the logical operators are "not" (`~`) or (`.not.`), "and" (`&`) or (`.and.`), and "or" (`|`) or (`.or.`). The operands of `&` and `|` must be of logical type. Both `|` and `&` associate from the left.

Operators	**Precedence**
(subscripting, reference)	9 (highest)
**	8
* *! / ! // /!	7
+ −	6
:	5
= == ~= < > <= < >= >	4
~	3
&	2
|	1 (lowest)

15.4 Delimiters

The delimiters used in expressions are parentheses `(, )`, brackets `[, ]`, comma `,`, and colon `:`. We have given a few examples where parentheses were used to change (or emphasize) the order of operations. In order to understand the function of parentheses in expressions, consider first the rule for evaluating expressions without parentheses:

Evaluate operations in order of precedence, highest first. When there are multiple operations with the same precedence, evaluate the expression from left to right (except for **, which is evaluated from right to left).

When an expression contains parentheses, add the following rule:

Evaluate inside the most deeply nested set of parentheses first, then move outwards through the successive levels of nesting. Thus, parentheses can be thought of as operators that raise the precedence of operators enclosed within them to a higher value than those at any lesser nesting level.

For example, the expression

```
a*b + c*d
```

is perfectly legal, but both multiplies are performed before the addition. To force the addition to be evaluated first, rewrite the expression as

```
a*(b + c)*d.
```

As mentioned in the section on predefined and user-defined functions, parentheses are used to delimit the actual arguments of these functions. Thus,

```
sqrt(2.0*18)
```

returns 6.0, and

```
mod(17,5)
```

returns 2.

The actual arguments of a function can be any expression that evaluates to a meaningful type (one cannot extract the square root of a logical, for instance). Naturally, function references themselves can occur in actual arguments, as in

```
sqrt(sqrt(3 + mod(17,2))).
```

Finally, parentheses can be used to delimit subscript and subscript-range references for subscripted variables. Individual elements of an array are themselves scalars, and can be accessed by specifying a list of expressions in parentheses separated by commas. The number of subscript expressions specified must be less than or equal to the number of subscripts declared for the variable, and the values must be in the proper range. For instance, if we declare

```
INTEGER x(3:10), y(-5:1,6)
```

then `x(4)` and `y(1,1)` are legal references to elements of these arrays. A reference to `x(1)` is illegal because the subscript is out of range; `y(3,5,1)` is illegal because there are too many subscripts. If fewer subscripts are given than are declared for the variable, the unassigned elements (to the right) default to their minimum legal value. Subscript expressions, if not integer, are converted to integers upon evaluation.

In addition to references to single elements of an array, references to the entire array or to certain portions of it are allowed. This is discussed fully in the next section.

15.5 Array References and Operations

15.5.1 Subscript References

Any operand in an expression may be a reference to an entire array or to a non-scalar subset of it. In such a reference, the name of the array may be given alone, or followed by subscript specifications separated by commas. If subscripts are not present, then the entire array is taken to be the operand. When subscripts are present, the number of subscripts must be less than or equal to the dimensionality of the named array. Subscripts can be one of the following:

- Nothing The default low and high subscripts are used. These are the actual limits for that subscript.
- An integer Any expression that evaluates to a scalar. The scalar is converted to an integer if necessary. This subscript refers to a single entry.
- A range A range is specified by low:high or low:high:increment, where low, high, and increment (if present) are each any expressions that evaluate to scalars, or nothing. If low and/or high is omitted, the actual limit for a subscript is used. If increment is not present it defaults to 1. Zero is an illegal value for increment, but negative values for increment are legal. Expressions are converted to integer if necessary, and high must be greater than or equal to low (unless of course increment is < 0, in which case low must be greater than or equal to high).
- A vector of integers An arbitrary one-dimensional array of integers is allowed as a subscript of a one-dimensional array of numeric type. Naturally each element of the subscript array must be within the range of subscripts of the array being subscripted. If `x` is an array of variables and `i` is an array of subscripts, then `x (i)` is an array the same length as `i` whose entries are `x (i (1))`, `x (i (2))`, `x (i (3))`, ... `x (i)` can be a component of an expression or the object of an assignment. In the latter case, if there are repetitions in `i`, then the order of assignment is undefined.

The Basis Language has the unusual property that the user may subscript expressions, not just variable names. Subscripting has the highest possible precedence and multiple sets of subscripts are evaluated left to right. For example,

```
(x-y)(3:5)
```

is the vector `[x(3)-y(3),x(4)-y(4),x(5)-y(5)]`. In an expression, the lowest subscript of the expression is the common lowest subscript of the operands, if they agree, and 1 if they do not.

15.5.2 Dimensionality

Each array has a shape, expressed as a dimension n (0 to 7) and a string of n integers (i1, i2, ..., in) representing the length of the array in each dimension. When a variable is used in an expression,

the resulting object, after applying the subscripts, may have some of its dimensional lengths equal to 1. Each such component is dropped and the dimension of the object reduced accordingly. Thus, `x(5)` is a scalar (dimension = 0) and `y(3:7,6,2:5)` has dimension 2 and shape (5,4).

All operands in an array expression must be the same size and shape, or else be scalars. Basis automatically creates an object of the appropriate size and shape from any scalar in the expression. Thus, for instance

```
a(1:3,2:5) + 2
```

adds 2 to each element of an array whose first subscript is 1, 2, or 3 and whose second element is 2, 3, 4, or 5. On the other hand,

```
a(1:3,2:5) + b(1:3,2:4)
```

is illegal because the two sizes cannot be made to conform;

```
x(1:6) + a(1:2,1:3)
```

is illegal because the shapes (i.e., number of dimensions) are different.

When ordinary scalar operators, such as `*, /, +`, and `-`, are used among objects of the same size and shape, they represent component-by- operator. Thus,

```
a * b
```

multiplies the matrices `a` and `b` component-by-component. This is not matrix multiplication, for which there is a separate operator (See “Array Operators” on page 77.)

15.5.3 Subscripts on Basis-created Variables

Basis constructs variables for you in several cases:

1. An assignment is made to a non-subscripted chameleon variable.
2. An assignment is made to a non-subscripted variable, which doesn’t exist, and `autovar=yes`.
3. A function is called with arguments and the formal parameters must be created to contain the actual parameters.
4. A result is printed and `$` must be created to “remember” it.

In all of these cases, the new variable's lowest subscript in each dimension is the same as that of the item being assigned to it. The highest subscript is the lowest subscript minus one plus the length in that dimension.

Basis also creates temporary values during the computation of expressions. These have a lower subscript, a high subscript, and a stride that is used for labeling printed results. When an operation takes place, if all parties to the operation agree about things, the result continues to be of the same shape. (Scalars that are broadcast are treated as agreeing.) If the parties to the operation differ in strides, the result has stride 1. If they differ in lower subscripts, the result has lower subscript 1. These rules are applied on a per-dimension basis.

The built-in function `fromone` can be used to force lower subscripts and strides to 1.

15.5.4 The Square Bracket Operator

The square bracket operator can be used to build arrays. On the simplest level,

```
[3,4,5]
```

is a one-dimensional array whose contents are 3, 4, and 5. The following bracketed subscripts

```
[[1,2], [3,4], [5,6]]
```

represent a two-dimensional array whose contents are

```
1        3        5
2        4        6
```

Note that `[1,2]` is the first column, not the first row. (This was done for compatibility with Fortran, which stores arrays in column-major order). Expressions can appear in the array-builder brackets. For instance, given the declaration

```
INTEGER a(1:3, 1:3)
```

then the array expression

```
[a(,1), a(,2), a(,3)]
```

is exactly the same as `a`.

As another example, suppose that i is declared as follows:

```
INTEGER i = [22, 3, 45, 23, 2, 56]
```

Then if `x` is a one-dimensional array, `x (i)` is exactly the same as:

```
[x (22), x (3), x (45), x (23), x (2), x (56)]
```

If different arguments to the square bracket operator have different types, the result is formed by coercing all elements of an array to the same type in the usual hierarchy: *integer* → *real* → *complex*. Thus,

```
[1,2,5]   is integer,
[2,3,5.]  is real, and
[2,3,5i]  is complex.
```

The square bracket operator can also take a sequence of operands which are not all the same size and shape. The operator consumes its operands from left to right. At each stage, then, there are two operands, the result so far (call it s) and the next operand (t). First, s and t are coerced to the same type. Then, if s has zero length, the result is t. Otherwise, if t has zero length, then the result is s. Finally, assume that neither s nor t has zero length, and that ns and nt are the dimensions of s and t.

Let n = min(ns,nt), and let m be the largest dimension such that the size of s and t match in the first m dimensions. The result will have dimension m+1. The length of the m+1 direction will be the sum of the lengths of s and t considering them as arrays of dimension m+1 with the first m dimensions equal to their current value.

Thus, if s has shape (3,5,6) and t has shape (3,5,12) the result is of shape (3,5,18). If t has instead shape (3,4) then the result has shape (3,5*6+4) or (3,34). If t was a vector of length 3, the result would be of shape (3,5*6+1), since thinking of t as an array of dimension two its shape is (3,1).

This definition of the square bracket operator reduces to the correct result for the simple case when all the arguments to the operator are of the same size and shape. The square bracket operator always has a defined result as long as its arguments can be coerced to a common type.

15.5.5 Array Operators

The four array operators are "matrix multiply" (`*!`), "matrix divide" (`/!`), "transpose" (`transpose(x)`), and "dot product" (`!`) or (`.dot.`). (In a previous version of Basis, transpose was an operator; now it is a function; but we leave its description here for easy reference.)

The matrix multiply, matrix divide, and transpose operators are peculiar to arrays of two dimensions; they cannot be used in other contexts. Also, the matrix multiply operation `*!` must be distinguished from that performed by `*` written between two matrices, which simply multiplies the corresponding elements of the two matrices.

Two matrices multiplied with `*!` must have the property that the number of rows of the first equals the number of columns of the second (but the second can be one-dimensional and thought of as a column vector).

The result of the matrix divide operation

```
b /! a
```

is the solution x to the equation

```
a *! x = b
```

so that a*! (b /! a) is b. If a is singular, b /! a is an error. The numerator b may be a vector or a matrix; the result is of the same shape. If a is of type integer it is coerced to type real.

The transpose function transpose(x) exchanges the rows and columns of its operand. For example,

```
transpose(a(1:3, 2:7))
```

results in a matrix whose shape is (2:7, 1:3) and whose elements (i,j) contain the values that were in a (j,i).

The final array operator is !, the dot product operator, e.g.,

```
[1,2,3] ! [0,1,4] = 14.
```

The dot product can be applied between any two objects whose sizes are equal, regardless of shape. For example,

```
[[2,3],[4,5]]![1,2,3,4] = 2*1 + 3*2 + 4*3 +5*4 = 40.
```

Non-arithmetic operators can be used with array operands. &, |, and ~ can be applied to compatible arrays whose entries are logical values (i.e., true or false). The operations of = (==) and ~= (<>) can be applied between pairs of arrays of compatible size and shape whose elements are of any type. The remaining relational operators such as > can be applied between real and integer arrays.

In all these cases the result is a logical array. The built-in functions land and lor can be used to reduce logical arrays to the single logical value required in IF tests.

An important thing to emphasize again about size and shape is that any object with a single-value subscript range is an object of fewer dimensions. For example,

```
INTEGER x(1), y(1,5), z(5,1)
```

declares a vector x and matrices y and z; but when used in expressions, x is a scalar and y and z are vectors, not 1 x 5 or 5 x 1 matrices. Likewise the matrix product of a matrix and a vector is a vector, not an n x 1 matrix. Thus, if the declaration

```
INTEGER x(5,5), y(5)
```

is followed by

```
$a = x *! y
```

this implies that `$a` is a vector with 5 elements, not a 5 x 1 matrix.

15.6 The Concatenation Operator

The `//` operator has the same precedence as `*`, `*!`, `/`, `!`, and `/!`. It is called the concatenation operator and is defined in three cases: (1) both arguments equal to scalar character strings, (2) both arguments arrays or scalars of type logical, and (3) both arguments arrays or scalars of type(s) integer, real, double, or complex. The usual coercion rules apply in the latter case if the arguments have differing types.

15.6.1 Concatenating Character Strings

When its operands are scalar character strings, `//` simply performs string concatenation. For example, after:

```
$a="en"
$b="dow"
$c=$a//$b//"ment"
```

the variable `$c` will have the value “endowment”.

15.6.2 Concatenating Numerical and Logical Arguments

Two logical or numerical objects of any size or shape may be concatenated, producing a one-dimensional array whose total number of elements is the sum of the numbers of elements in the two concatenated objects, in the order in which they occur in memory (which means column- does things). For example:

```
$a=3//4
```

produces the vector `[3,4]`, while

```
$b=[[2,3],[4,5]]//[6,7]
```

results in `[2,3,4,5,6,7]`. If one wanted to produce a 2 by 3 matrix from this, which has `[6,7]` as its last column, one could use the shape operator, thus:

```
$b=shape($b,2,3)
```

forcing the concatenated result into the desired shape.

As a second example, consider the code fragment below. The routine "update" accepts the incoming values of the array `y` and the scalar `t` and returns new values. These new values are concatenated onto an accumulation of the older values, and then forced into a shape such that they can be displayed in rows with `t` in the first column and the corresponding `y`'s in columns 2, 3, and 4.

```
real y(3) = [1., 2., 3.], t = 0.
integer i
$a = t // y                        #initialize output
do i = 1, 10
     call update (&y, &t)          #note call by reference
     $a = $a // t // y             #add next solution
     t = t + 0.1
enddo
$a = transpose(shape($a, 4, 11))
```

Note the use of the transpose operator in the last expression. If this were not used, we would see t, then `y(1)`, `y(2)`, `y(3)`, etc., running down the columns if we printed `$a` out, instead of across the rows.

Logical arrays and scalars whose entries are logical values ("true", "false") may be concatenated following the rules above. Logical and numeric objects are incompatible and cannot be mixed in concatenations. Neither argument of a concatenation may be a structure, even if all of its entries are numeric. The result of such an operation, if it is attempted, will be unpredictable.

CHAPTER

SIXTEEN

Display and Assignment Statements

A display statement is simply a list of expressions separated by commas. When a display statement executes, the expressions are evaluated left to right, assigned to the special chameleon variable `$`, and then displayed. At the end of execution, `$` has the value of the last expression computed For example,

```
3 + 1, 2
```

will display 4, then 2, and the variable `$` will have the value 2 at this point. If a semantic error occurs during the execution of a display statement, execution of the remainder of the statement is aborted, and `$` has the value of the last correct expression evaluated.

The variable `autohist`, 42.1, can be used to cycle the results through `$a`, `$b`,...,`$z` instead of always using `$`.

Note that only arithmetic and string-valued expressions, i.e., <`exp`>s, can be displayed in this way. The syntax of the display statement does not allow for the full generality of <`lexp`>s, which include logical-valued expressions. However, this limitation can be circumvented, and the values of logical expressions displayed, by placing parentheses around them, thus:

```
(x + y < 2)
```

Without this restriction we would be unable to translate the statement `a = b` because we could not tell if this is an assignment statement or a display of a logical expression.

The assignment statement has the general form

```
target = source,
```

where `source` is an <`lexp`> as described in the last section, (i.e., any expression, capable of evaluation, of any type, size, or shape). The target is the object where the value or values of the source object will be stored. If `target` is a scalar or a scalar element of an array, then the statement is a simple scalar assignment and needs no further explanation.

If `target` is a chameleon variable it assumes all the characteristics of `source` (size, shape, and type), and then receives the value(s) of the source object. It is not possible to generate an error

when an unsubscripted chameleon variable is the target object, unless the variable does not exist (i.e., has not been declared).

Array assignments are more difficult. Generally, if the target and the source expression are not of the same shape, it must be possible to store the expression as a subset of the target object. However, if the target is an array and the source is a scalar, then the scalar value will be broadcast to all specified elements of the array.

If the number of subscripts given in the assignment statement,

```
variable(subscripts) = expression
```

is less than the actual dimension of variable, it is assumed that the remaining subscripts have their lowest value (typically 1). If no subscripts are given in the assignment statement, the target is the full array variable. The shape of the target object may contain some 1's. The true shape of the target is its shape with the 1's dropped. There are two conditions on the true shape of the target:

- It must be of at least as many dimensions as expression.
- Each component of the target must be at least as large as the corresponding component of the expression.

If both these conditions hold, then `expression` can be stored as a sub-object of `variable`. Here are some examples:

If `x` has shape (3,2) then

```
x(2) = 5    sets x(2,1) to 5
x(2,) = [5,6]   sets the second row of x to [5,6]
x(2:,) = [[5,6],[7,8]]  sets the 2 by 2 submatrix of x whose upper
left corner is x(2,2) to the matrix

% MathFF:matrix[2,2,num[5.00000000,"5"],num[7.00000000,"7"],
% num[6.00000000,"6"],num[8.00000000,
% "8"]]

x(1:3:2,) = [[5,6],[7,8]]   sets the 2 by 2 submatrix of x
consisting of rows 1 and 3 and columns 1 and 2 to the matrix

% MathFF:matrix[2,2,num[5.00000000,"5"],num[7.00000000,"7"],
% num[6.00000000,"6"],num[8.00000000,
% "8"]]
```

The assignment `x(2,) = x(,2)` is erroneous: `x(,2)` has shape (3) while `x(2,)` has shape (2). If `x` had been a square two-dimensional array, however, this would have correctly set the second row to the second column.

If `y` has shape (5,6,7) then

```
y(3,2:6,1:7) = x
```

is a correct assignment. The target has shape (1,5,7). Its true shape is (5,7). The source has shape (3,2), which is smaller in each component. The assignment is performed beginning at `y(3,2,1) = x(1,1)`, `y(3,3,1) = x(2,1)`, `y(3,4,1) = x(3,1)`, `y(3,2,2) = x(1,2)`, etc.

Assignment is allowed to a one dimensional array subscripted by an arbitrary subscript array, e. g.

```
x ( [20, 13, 3, 56, 43, 5]) = y (3:9)
```

assigns `y (3)` to `x (20)`, `y (4)` to `x (13)`, `y (5)` to `x (3)`, etc. Note, however, that the result of an assignment to a variable with repeated subscripts, such as

```
x ( [20, 13, 3, 13, 43, 13]) = y (3:9)
```

is undefined. This is because on some architectures this assignment will be parallelized, in which case we do not know the order in which the assignments to `x (13)` will occur.

One special case requires some thought to understand. As an assignment target, `x` and `x()` are very different. In the latter case, one subscript has been given, although defaulted, and hence that one subscript defaults to its lowest possible value; and any other subscripts will then default to their lowest possible values, since they were omitted. Thus the target `x()` is the first element of `x`, while the target `x` is all of `x`.

16.1 Assignment Actions

For each variable, the user may specify a string containing Basis language statements called its assignment-action string. This string will be parsed and executed after each assignment statement in which the corresponding variable name appears on the left-hand side of the assignment statement. See 43.12.

16.2 Operator Assignments

The form of an operator assignment statement is

```
target op= source
```

where `op` can be any of the seven operators `+`, `-`, `*`, `/`, `|`, &, or `**`. Many readers are no doubt familiar with the operator assignments from C and C++. The above statement has the same effect as

```
target = target op source
```

and so it can be thought of as a shorthand notation. For example,

```
i = i + 1
```

can be written with fewer keystrokes as

```
i += 1
```

This is especially handy if the left-hand side of the assignment is a long identifier with many subscripts.

The left side of an operator assignment can be a one dimensional array with an arbitrary array of integer subscripts. However, if any of the subscripts is repeated, then the resulting element with that subscript is not defined.

16.3 The Append Statement

The append statement is part of a facility in Basis which assists in the process of collecting lists of values, such as time histories, in an efficient manner. The components of the facility are:

- `setlast`, a routine for imposing a limit on the last subscript.
- The `:=` "append" operator.
- `rtadddim`, a routine for "adding" a dimension to a variable.

The routine `setlast(`*`name, n`*`)` limits the LAST dimension (only) of the variable name to length of *n*. If *n* is greater than the current length (unlimited) of last subscript of name, then an attempt is made to expand storage so that the length will be at least *n*.

If *n* is greater than the current maximum value, then the maximum is set to 1.5 times the existing value or at least 16. This exponential growth is used to help reduce memory fragmentation while preserving constant time operation cost. `setlast` can be used on static arrays as long as no attempt is made to exceed the actual storage available.

The append operator `:=` works as follows: `x := y` is equivalent to storing `y` after the current end of `x`, increasing the final subscript of `x` appropriately (using `setlast`'s internal routine `rtsetdl`). `y` must be of an appropriate shape to be so stored. If `y` is of the same dimension as `x`, `y` is viewed as an array of values to be added, and the final subscript of `x` will increase appropriately. If `x` is a scalar, it is first made a one-dimensional vector of length 1.

`rtadddim`("name") adds a new subscript of 1 to name. This can be useful in setting name up as a target for a `:=`.

Example 1:

```
real x(3,3)
call setlast("x", 2)  # x will act as if it is shaped (3,2)
```

Example 2:

```
real x(3,3)
call setlast("x", 0)  #x will act as if it is shaped (3,0)
x:=iota(3)   # now x is (3,1) (but storage is still (3,3) )
x:=iota(3)   # now x is (3,2) (but storage is still (3,3) )
x:=iota(3)   # now x is (3,3) (but storage is still (3,3) )
x:=iota(3)   # now x is (3,4) (but storage is now   (3,16) )
x:=iota(3)   # now x is (3,5) (but storage is still (3,16) )
x:=[iota(3),iota(3)]
             # now x is (3,7) (but storage is still (3,16) )
```

Example 3:

```
integer y(0) # set up an empty array
integer i
do i=1, 1000
   y:=i
enddo
# After this loop, y is the same as iota(1000)
```

16.4 The Logical IF Statement

The IF statement in Basis takes two forms that are similar to the Fortran logical IF and block IF statements. We use this same Fortran terminology when referring to the two IF statements in Basis.

The syntax for the logical IF is

```
IF (<lexp>) <nonnullstatement>
```

where semantically <lexp> must evaluate to a scalar logical value. The <nonnullstatement> can, but need not, be on the same line as the IF (<lexp>). Furthermore, unlike Fortran, the only restriction on the type of controlled statement is that it cannot be null. Thus, in principle, logical IFs (and other structured statements) can be nested to any depth.

In the following example,

```
IF (a < b & b < c) m = c
```

sets `m` precisely to `c` if both `a < b` and `b < c` are true. A more complicated example is

```
IF (i <= maxindex)
     IF (a(i) ~= 0)
          b(i) = b(i) / a(i)
```

The nested logical IFs illustrated above are not equivalent to the single IF statement

```
IF(i<=maxindex & a(i)~=0) b(i) = b(i)/a(i)
```

because in the evaluation of a conjunction (expression with `&`), both operands of the conjunction are always evaluated even if the first operand is false. Thus, if `i > maxindex`, an attempt would still be made to evaluate `a(i)`, which would cause a semantic error at run-time (subscript out of range). For this reason, the first form is preferred.

Like all Basis statements, IF statements are actually compiled into a low-level code, and this code is not interpreted (i.e., executed) until the complete statement has been read in. If errors in syntax (i.e., the grammatical form of the statement) are detected during the compilation process, compilation is aborted and the offending statement must be retyped. Once a statement is entered correctly, it executes to completion unless the detection of a semantic error aborts execution. If execution was nested inside one or more structured statements, user functions, or both, when the error occurred, information about the nesting is displayed.

The normal Basis prompt at initialization is[1]:

```
Basis>
```

During the input of structured statements, however, this prompt changes to a series of $>$ symbols that indicate the nesting level. The prompt returns to normal after execution completes. For instance, the example above with Basis prompts is:

```
Basis > IF (i <= maxindex)
\>              IF (a(i) ~= 0)
>>                   b(i) = b(i)/a(i)
Basis >
```

16.5 The Structured IF Statement

The other type of IF statement is quite similar to the Fortran block IF. In skeletal form, it looks like this:

[1] In an application code the main prompt is usually changed by the author

```
IF(<lexp>) THEN
   <stlist>
ELSEIF (<lexp>) THEN
   <stlist>
...
     ELSE
   <stlist>
ENDIF
```

where <stlist> represents either a single statement or a sequence of statements separated by semicolons or carriage returns.

ELSEIF and ENDIF must be single words. If you enter `ELSE IF`, for instance, then the compiler will think that a new IF statement, nested inside the current one, is being started. `END IF` will cause compilation to abort with a syntax error. The ELSEIF clause is optional. Also note that the ellipsis above indicates that there may be many ELSEIF clauses. The ELSE clause is optional also, but, of course, there can be no more than one ELSE clause.

Basis is not overly particular about the placement of THEN; it can be on a separate line, and it can, but need not, be followed by a statement on the same line. In fact, THEN can be omitted from an ELSEIF clause, provided that <stlist> begins with a non-null statement. Thus, although THEN is not syntactically important, ENDIF, ELSEIF, and ELSE are. ENDIF, ELSE, and ELSEIF must each appear at the beginning of a separate line, or be separated from <stlist> by a semicolon.

Here is a block IF that determines the maximum of two numbers:

```
IF (a>b) THEN
     m = a
ELSE
     m = b
ENDIF
```

This could equally well be written

```
IF (a>b)
THEN m = a
ELSE m = b
ENDIF
```

or could even be written on one line as

```
IF (a>b) THEN m = a; ELSE m = b; ENDIF
```

Semicolons are required to separate statements that appear on a single line.

The following nested block IFs determine the maximum of three numbers:

```
IF ( a>b ) THEN
     IF (c>a) THEN
          m = c
     ELSE
          m = a
     ENDIF
ELSEIF (c>b) THEN
     m = c
ELSE
     m = b
ENDIF
```

Of course, the built-in function max(a,b,c) is a lot easier!

CHAPTER

SEVENTEEN

WHILE Statement

17.1 WHILE Statement

The WHILE statement is a loopingconstruct similar to that found in C:

```
WHILE (<lexp>)
     <stlist>
ENDWHILE
```

As in the IF statement, <lexp> is required to evaluate to a logical value (true, false). Otherwise a semantic error occurs and execution is terminated. At execution time, <lexp> is evaluated. If it is false, execution of the WHILE loop is terminated. If the WHILE loop is nested inside another statement, then control goes to whatever follows the ENDWHILE. If <lexp> is true, then <stlist> is executed. If <stlist> does not contain a NEXT, BREAK, or RETURN statement in its flow of control, then <lexp> is evaluated again and execution proceeds as above. (The NEXT, BREAK, and RETURN statements alter the flow of control and may cause the loop to terminate.)

For example, the following sequence of statements adds up the positive elements in array a. Note the IF statement nested within the WHILE.

```
sumpos = 0
i = 1
WHILE (i<=amax)
     IF (a(i)>0) sumpos = sumpos + a(i)
     i = i + 1
ENDWHILE
```

Below is a set of nested loops that compute the product of two square matrices a and b and place the result in c:

```
i = 1
WHILE (i<=n)
```

```
     j = 1
     WHILE (j<=n)
          c(i,j) = 0
          k = 1
          WHILE (k<=n)
               c(i,j) =  c(i,j) + a(i,k)*b(k,j)
               k = k + 1
          ENDWHILE
          j = j + 1
     ENDWHILE
     i = i + 1
ENDWHILE
```

The Basis prompt on the innermost loop will be >>>.

17.2 BREAK and NEXT Statements

The BREAK statement provides a way to exit from a loop other than via the controlling condition going false. Indeed, a WHILE (true) will never terminate unless it contains a BREAK. For example, the following example is equivalent to the first WHILE in the first example given above:

```
sumpos = 0
i = 1
WHILE (true)
     IF (i<=amax) THEN
          IF(a(i)>0)sumpos = sumpos + a(i)
          i = i + 1
     ELSE
          BREAK
     ENDIF
ENDWHILE
```

Normally, as is the case here, the BREAK statement will be controlled by some sort of test.

BREAK can be used to exit from nested structures by using the form BREAK n (or BREAK(n)), where n is the level of loop nesting. BREAK and BREAK 1 mean the same thing.

Here is an example using BREAK from a nesting level of three deep:

```
i = 1
WHILE(true); j = 1
     WHILE (j<=5); k = 1
          WHILE(k<=5)
               IF(i*j*k > 86) BREAK 3
```

```
            k = k + 1
        ENDWHILE
        j = j + 1
    ENDWHILE
    i = i + 1
ENDWHILE
```

The nesting level expressed in a BREAK can, but need not, be enclosed in parentheses. It must be a positive integer constant, however, not an expression or variable name.

The BREAK statement is not used exclusively in WHILE statements; it can be used inside any iterative statement (iterative statements are discussed later). Bear in mind that a specified nesting level is the level of nesting inside loops only; the fact that each BREAK in the examples above is in an IF does not affect its level as far as loops are concerned. A BREAK statement that occurs outside a loop, or one that occurs inside a loop with an expressed nesting level greater than the actual nesting level, will simply be ignored and has no effect whatsoever.

The NEXT statement has a provision for prematurely reentering a loop (including an outer loop that contains the loop with the NEXT statement). For example, the following loop adds all the elements of array `a`, except those whose subscript is divisible by 5:

```
i = 0; sum = 0
WHILE (i<n)
    i = i + 1
    IF(i/5*5 = i) NEXT
    sum = sum + a(i)
ENDWHILE
```

The integer `i/5*5` is equal to `i` precisely if `i` is divisible by 5, in which case NEXT causes control to return to the top of the loop, where `i < n` is checked again.

In the NEXT statement, as in BREAK, an optional nesting level can be given, either as an absolute integer or as an integer in parentheses. NEXT and NEXT 1 are equivalent. NEXT 2 causes iteration to proceed to the top of the next outer loop, and so on. As with BREAK, a NEXT is ignored if its expressed nesting level is greater than the current actual level, or if it occurs outside a loop.

CHAPTER

EIGHTEEN

FOR Statement

The FOR statement, except for minor syntactic variations, is similar to the one in the C Language; C programmers should beware the interchanged roles of comma and semicolon.

Its general form is

```
FOR (<forinit>, <lexp>, <stlist2>)
     <stlist1>
ENDFOR
```

where <forinit> is a (possibly null) list of one or more assignment statements, separated by semicolons or carriage returns. These initializations are performed exactly once, when the loop is first entered from above. The logical expression <lexp>controls iteration. If it evaluates to false, the loop is exited; and if it is true, <stlist1> executes, then <stlist2>, and then <lexp> is tested again, etc. BREAK works exactly as described in the preceding section, while NEXT transfers control to the execution of <stlist2>.

Logically, the FOR loop above is equivalent to

```
<forinit>
WHILE (<lexp>)
     <stlist1>
     <stlist2>
ENDWHILE
```

except that NEXT transfers control to <stlist2>.

Normally <forinit> might be used to initialize a loop control variable, <lexp> to test it, and <stlist2> to increment it at the end of the loop. For example

```
sum = 0
FOR(i = 1, i<=n, i = i + 1)
     sum = sum + a(i)
ENDFOR
```

adds up the elements of array `a`. This could also be written as

```
FOR (i = 1; sum = 0, i<=n, sum = sum + a(i); i = i + 1)
ENDFOR
```

or even

```
FOR (i = 1
     sum = 0,
     i<=n,
     sum = sum + a(i)
     i = i + 1)
ENDFOR
```

The semicolons separating statements are not required if statements are on separate lines. However, the commas between the three parts of the FOR header are required.

Note that ENDFOR must be preceded by a semicolon or carriage return.

Here is a matrix multiply using a FOR:

```
FOR (i = 1, i<=n, i = i + 1)
     FOR (j = 1, j<=n, j = j + 1)
          a(i,j) = 0
          FOR (k = 1, k<=n, k = k + 1)
               c(i,j) = c(i,j) + a(i,k)*b(k,j)
          ENDFOR
     ENDFOR

ENDFOR
```

CHAPTER

NINETEEN

DO Statement

19.1 Uncontrolled DO

There are three forms of the DO statement. The first, called for obvious reasons the uncontrolled DO, is the simplest in form:

```
DO
     <stlist>
ENDDO
```

The initial DO must be followed by, and the ENDDO preceded by, a semicolon or carriage return, as indicated above. In this uncontrolled DO, <stlist> repeatedly executes; in fact, if it does not contain a BREAK statement, or if it does and it never executes, then the loop repeats forever.

19.2 DO-UNTIL

The second type of DO, DO-UNTIL always executes its body once, and performs the test at the end.

```
DO
     <stlist>
UNTIL (<lexp>)
```

In the DO-UNTIL loop, <stlist> executes and then <lexp> is tested. If <lexp> is true, the loop terminates; if it is false, <stlist> repeats, and so on. The logical expression must evaluate to a logical scalar at run-time, or a semantic error will occur.

BREAK works exactly as it does in other types of loop to effect exit. NEXT works in a reasonable way, but maybe not as one might expect without a little thought. NEXT causes control to proceed directly to the top of <stlist>, bypassing the UNTIL (<lexp>, on the philosophy that when reinitiated, this type of loop always executes its body once before testing at the end.

19.3 Controlled DO

The third type of DO is appropriately called the controlled DO, because its iterations and termination are controlled by an explicitly named scalar whose initial and termination values (and optionally, increment) are specified prior to execution of the loop. This construct is quite similar to the one from Fortran:

```
DO <lhs> = <init>, <term>, <incr>
     <stlist>
ENDDO
```

The controlling scalar <lhs> must be integer and scalar; unlike FORTRAN, it may be an element of an array. The other loop specifications, <init>, <term>, and <incr> must be scalar expressions with numeric values which can be coerced to integer. The ", <incr>" can be omitted. If it is, it defaults to 1 as in Fortran.

The controlled DO is roughly equivalent to the following statements, where %C1 and %C2 can be thought of as variables accessible only to the Basis run-time system that cannot be changed by the user:

```
     <id> = <init>
     %C1  = <term>
     %C2  = <incr>           # Or 1, if <incr> is absent
DO
     IF (%C2 > 0 & <id> > %C1) BREAK
     IF (%C2 < 0 & <id> < %C1) BREAK
     <stlist>
     <id> = <id> + %C2
ENDDO
```

Thus, the control expressions <term> and <incr>are evaluated exactly once, when the loop is entered. This is important because it enforces the concept that the loop will execute a number of times that is known upon entry, and that the number of iterations will not change subsequently, even if the user alters components of the expressions <term> and <incr>. Likewise, if the loop controlling scalar is a subscripted variable, its subscripts are evaluated exactly once, before the loop is entered. Even should these subscripts change within the loop, the same array element will still be used for the loop control. Finally, note that the test for loop exit is at the top of the loop, and that the incrementing is at the bottom (after each execution of the body).

The next DO loop squeezes the zeroes out of an array:

```
j = 1
DO i = 1, maxa
     IF(a(i) = 0) THEN
          maxa = maxa-1
```

```
     ELSE
          a(j) = a(i)
          j = j + 1
     ENDIF
ENDDO
```

The loop executes as many times as there were elements in a at the time the loop was entered, because the initial value of `maxa` is saved and used for loop control. Upon exit from the loop, `maxa` will have been changed to reflect the size of the smaller array.

The following example is, again, the matrix multiply; this time it is performed using several different DO loops:

```
i = 1
DO
     IF (i>n) BREAK
     DO j = n, 1, -1     # note negative increment
          c(i,j) = 0
          k = 1
          DO
               c(i,j) = c(i,j) + a(i,k)*b(k,j)
               k = k + 1
          UNTIL (k>n)
     ENDDO
     i = i + 1
ENDDO
```

CHAPTER

TWENTY

Functions Listed by Type

Basis contains three different kinds of functions: user-defined, built-in, and compiled. The latter two must be distinguished because there are different rules for using built-in and compiled functions. The built-in routines are documented in the following section. The compiled functions are documented in Chapter 43. The following tables are intended for browsing to locate the routine you need. They list the built-in and compiled functions classified by their general function or nature. Compiled functions are listed in *italic* face.

20.1 Common Mathematical

abs	aint	anint	exp	log	log10
alog	alog10	nint	ranf	sign	sqrt
mod	min	max	sup	inf	

20.2 Trigonometry

acos	asin	atan	atan2	cos	cosh
cot	sin	sinh	tan	tanh	

20.3 Type Conversion and Complex Numbers

aimag	cmplx	conjg	dble	float	int
sngl	struct				

20.4 Arrays

Most of the built-in functions can take arrays as arguments or produce them as output. These functions are helpful in working with arrays:

ave	cumaddin	diag	iota	land	lor
length	load	max	min	mnx	mxx
ones	outer	psum	ptp	ranf	rangex
rsum	shape	struct	sum	sup	inf
setlimit	*setshape*	where	gather	fromone	trueshape
truerange	setact	spanl	rmsdv	squeeze	setlast
rtadddim	*sorti*	trans-pose			

20.5 Character Manipulation

len_trim	index	trim	triml	trimr	substr
format	toupper	tolower			

20.6 Special Purpose

format	index	load	range	shape	struct
type	*help*	*news*	*dec*	*oct*	*hex*
allot	change	*basfree*	*gallot*	*gchange*	*gfree*
execuser	*comment*	*exists*	*flushlog*	*swset*	*switch*
protect	*paws*	*setmnarg*	*kaboom*	*parsestr*	setranf
getranf	seedranf	mixranf	cd,chdir	setenv	getenv
disk-	space				

20.7 Obtain/Set Scalar Values

ibasis	*rbasis*	*dbasis*	*cbasis*	*lbasis*	*sbasis*
sibasis	*srbasis*	*sdbasis*	*scbasis*	*slbasis*	*ssbasis*

CHAPTER

TWENTYONE

Built-in Functions

The user has access to a number of built-in functions, which are invoked in an expression by using the name of the function followed by a parenthesized list of its actual arguments. Basis can return not only scalars, but also array values or even what are called structures. A structure is an array whose individual elements can be objects of different types: scalars, arrays, or even other structures.

In this section, there is a brief alphabetic list of currently implemented built-in functions, their required parameters, and a description of what they return. Generally speaking, the built-in functions allow the user license in what arguments can be sent. The number of arguments is checked, but frequently the type of argument can be virtually anything, and a correct result will be returned. Most of the arithmetic functions, for instance, will accept arguments of any size and shape, apply the function to each component, and return an object of the same size and shape with the new components.

In what follows, unless otherwise noted, a single argument can be a scalar of type integer, real, double, or complex, or an array whose elements are of these types. A function is applied component by component to arrays. Unless otherwise noted, the result components are of the same type, unless they need to be coerced to real or complex.

This list can be obtained at run-time with the command "`list Builtin`".

abs(x) returns the absolute value of object x.

acos(x) returns the inverse cosine (in radians) of object x. The inverse trig functions do not return complex values (e.g., acos(2) = 0). If x is complex (or has complex components), acos is applied only to the real part(s) of x.

aimag(x) returns the imaginary part of object x. Use float to get the real part of complex x.

aint(x) returns the integer part of object x (i.e., truncates the fraction.) The result is always real. If x is complex, aint is applied to the real part of x.

alog(x) alog(x) = natural logarithm of x

alog10(x) alog10(x) = base 10 logarithm of x

anint(x) returns the nearest (real) whole number to x. See "aint(x)".

asin(x) is the inverse sin (in radians) of x. See “acos(x)”. 21

atan(x) is the inverse tangent (in radians) of x. See “acos(x)”. 21

atan2(y,x) is the inverse tangent (in radians) of the angle between the positive x axis and the vector whose components are (x,y). x and y can be vectors of the same length, or both scalars, or either can be a scalar and the other a vector. See “acos(x)”. 21

ave(x,idim) returns the average of array x. x can be of type complex, real, double or integer. The resulting type is the same as x, unless x is of type integer. In this case, the resulting type is real. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. The output array produced is the same shape as x, except that the i-th dimension is removed. If idim is not supplied, then the function is applied to the entire array, resulting in a scalar output.

cmplx(x), cmplx(x,y) returns x if x is complex; if x is integer or real, it returns a complex number with imaginary part = 0 and real part = x (componentwise if necessary). In the two argument form, returns the complex number (x,y). It is a semantic error if x and y are not real or integer, or if x and y are not the same shape (except that one could be a scalar, which would be broadcast).

conjg(x) returns the complex conjugate of object x.

cos(x) returns the cosine of object x, which must be in radians. x can be integer, real, double, or complex; cos(x) will be real, double or complex as necessary.

cosh(x) returns the hyperbolic cosine of object x. See “cos(x)”.

cot(x) returns the cotangent of object x. See “cos(x)”.

cross(x,y) returns the cross-product of two real 3-d vectors x,y.

cumaddin(&x(i),y `x` and `y` are one-dimensional arrays of numeric type, and `i` is an arbitrary integer array of subscripts into `x`. `i` and `y` are of the same length. Note that the ampersand on `x` is required, because the contents of `x` will be changed by this operation. The effect of this function is the same as the Basis loop do j = 1, shape (y) x (i (j)) += y (j) enddobut of course it is faster because the operations are done by compiled code. Note that if `i` contains repeated subscripts, then the effect of this is to accumulate the sum of corresponding values from `y` into the `x` values corresponding to the repeated subscripts. Also note that if `i` does not have repeated subscripts, then it is much easier to do this as x (i) += yThe latter will not work, however, if `i` has repeated values, because only one of the sums will be assigned, and furthermore, it is impossible to know which, if the computation is parallelized.

dble(x) returns an object of the same size and shape as x whose components are those of x, converted to double.

dcmplx(x [,y)]dcmplx(x) convert x to double complex type, dcmplx(x,y)=dble(x)+idble(y)

diag(x), diag(x,k) where x is a vector, returns a matrix whose main diagonal is x. The matrix will be n by n, where n is the length of x. diag(x,k), where x is a vector of length n and k is an integer, will return an `(n + |k|)` by `(n + |k|)` matrix with x on the kth diagonal (k may be negative; k = 0 is the main diagonal). The remainder of the entries are 0. If k is not an integer, it or its real part is truncated to integer. If k is not a scalar, its first component is extracted and used.

exp(x) returns the real, double or complex exponential of x, componentwise if necessary.

fft(x [,dim)] Fourier transform. x real or complex. If present, dim is the dimension over which the transform is taken for all values of the other subscripts. The transform length, n = length(x) or shape(x)(dim), can be any integer >0, but the method is most efficient when n is the product of small primes. For x complex, fft(x) returns z(j) = x .dot. exp(-2i*pi*j*iota (0,n-1)/n), j in range 0:n-1. For x real and n = 5 [6], fft(x) returns c0, c1, s1, c2, s2 [,c3], where cj = x .dot. cos(2*pi*j*iota(0,n-1)/n), and and sj = x .dot. -sin(2*pi*j*iota(0,n-1)/n). See also the inverse transform, ffti.

ffti(x [,dim)] Fourier inverse. For x real or complex, ffti(fft(x)) = x*length(x) for x one-dimensional, and ffti(fft(x,dim),dim) = x*shape(x)(dim) for any x with dimensionality >= dim. See also fft.

fit(x,y,n) fit(x,y,n) fits an n-th degree polynomial in x to y.

fromone(x) produces a value of the same type and shape as x, with its lowest subscript in each dimension set to 1.

float(x) returns an object of the same size and shape as x whose components are those of x, converted to real.

format(x,fw,nd,flg) returns a character string containing the formatted value of x. If fw is positive, the string length is fw; if fw is zero, the string has no leading blanks; if fw is negative, the string length is abs(fw) and the string is filled with leading zeros following the sign, if present. If nd is given, output is real with nd places after the decimal point. If flg is 1, output is fixed format; if 0, E- or D-format is used, depending on the type of x; if 2, E-format is used even if x is double precision

gather(x,index) gathers up a vector from source vector, x. x is a one dimensional array of type real, double, integer or complex. index is a one dimensional integer array which determines which elements are accessed. The output vector is the same type as the source vector. Its length is the same as the vector of indices. **EXAMPLE:**

```
real x(-2:2)=[-4,-2,0,2,4]
integer index(3)=[-2,0,2]
chameleon a=gather(x,index)
```

would result in:

```
a(1) = x(index(1)) = x(-2) = -4.00000e+00
a(2) = x(index(2)) = x(0)  =  0.
a(3) = x(index(3)) = x(2)  =  4.00000e+00
```

index(s,r) where `s` and `r` are strings, returns the position in `s` where `r` first appears as a substring, or zero if `r` is not found.

inf can have an arbitrary number of arguments of any sizes and shapes. inf returns a scalar which is the minimum value present amongst all the components of all the objects. Arguments must be of arithmetic type; only the real parts of complex objects participate.

int(x) returns x converted to integer, componentwise if necessary. If x is complex, int is applied to the real part. Conversion is by truncation.

iota(n), iota(m,n) where m and n are integer, returns a vector of length n-m, whose components, in order, are integers `m, m+1, m+2...,n`. If m is omitted it is assumed = 1.

land(x,y,...) can have an arbitrary number of logical arguments of any shapes. land returns a logical scalar which is true if every component of every argument is true.

length(a) returns the number of elements in a.

len_trim(s) returns the string length of string s without counting trailing blank characters.

load(a,n) returns a real vector with n components, the consecutive values in memory starting at address a. This function is useful for debugging.

log(x) returns the natural logarithm of object x, by components if necessary. See cos.

log10(x) returns the common logarithm of object x, by components if necessary. See cos.

lor(x,y,...) can have an arbitrary number of logical arguments of any shapes. lor returns a logical scalar which is true if some component of some argument is true.

max accepts two or more arguments and returns the maximum component by component. Scalars will be broadcast, but otherwise the arguments must have the same number of components. The result has the shape of the first non-scalar argument or is scalar if all the arguments are scalar. Only real parts of complex objects participate. See “sup”. 21

min does the same as max, but returns the minimum. See “inf”. 21

mnx(x,idim) returns the minimum index of array x. x can be of type real, double, or integer. The resulting type is the same as x. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. The output array produced is the same shape as x, except that the i-th dimension is removed. If idim is not supplied, then the function is applied to the entire array, resulting in a scalar output.

mod(x,y) returns the remainder after division of object x by object y. If x and y are not the same size and shape, y must be a scalar, which is then broadcast.

mxx(x,idim) returns the maximum index of array x. See “mnx(x,idim)”. 21

nint(x) returns the nearest integer to real object x. Similar to anint except for type of result.

ones(n) returns a vector of length n whose components are all 1, with a single scalar integer argument n. ones of more than one integer scalar argument returns an array of that shape whose components are the Kronecker delta.

outer(x,y) returns the outer product of objects x and y.

psum(x,idim) returns the partial sum of array x. x can be of type integer, real, double, or complex. The output is an array of the same type, size and shape as x. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. This value is stored in the output array element whose indices are the same as the indices of the input elements. If idim is not supplied, then the function is applied to the entire array.

ptp(x,idim) returns the peak to peak of array x (maximum value - minimum value). See “mnx(x,idim)”. 21

ranf(x) returns an object of the same size as x whose components are random numbers. These will be between 0 and 1 if x is integer, real, or double, on the unit circle if x is complex. See the chapter on Compiled Functions for additional documentation about ranf and its supporting routines *setranf*, *getranf*, *seedranf*, and *mixranf*.

rangex(x) where x is an array, returns a matrix whose rows contain the lower and upper subscripts for each dimension of x which is not of length 1. If x is scalar range, returns `[1,1]`.

EXAMPLES:

```
integer x(3,2:4,1,5), y(3,2:4,2,5)
rangex(x    # returns 4x2 matrix with rows
    # [1 3], [2,4], [1,1], [1,5]
rangex(y(,,,1:4))   # returns 4x2 matrix with rows
    # [1 3], [2,4], [1,2], [1,4]
```

rmsdv(x,idim) returns the root mean square deviation of array x. x can be of type real, double precision, or integer. The resulting type is the same as x, unless x is of type integer. In this case, the resulting type is real. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. The output array produced is the same shape as x, except that the i-th dimension is removed. If idim is not supplied, then the function is applied to the entire array.

rngbeg(rng, begindx) rng - the range (or array of ranges) whose beginning index (or indices) is to be returned.begindx - the integer value (values) to be used for any beginning index (or indices) whose value has been DEFAULTED.

This function returns the beginning index (or indices) of the given range(s) (argument rng) in which all the DEFAULTED fields (e.g. :10) have been replaced by the corresponding value

of argument begindx. If argument rng is an array, then argument begindx must be either a scalar integer or an array of the same length as argument rng.

EXAMPLES:

```
rngbeg(:8,2)    # returns 2
rngbeg(1:4, 3)  # returns 1
rngbeg([:8, 1:4, :7], 3)
    # returns vector [3,1,3]
rngbeg([:8, 1:4, :7], [1,2,3])
    # returns vector [1,1,3]
```

rngend(rng, endindx) rng - the range (or array of ranges) whose ending index (or indices) is to be returned.endindx - the integer value (values) to be used for any ending index (or indices) whose value has been DEFAULTED.

This function returns the ending index (or indices) of the given range(s) (argument rng) in which all the DEFAULTED fields (e.g. 2:) have been replaced by the corresponding value of argument endindx. If argument rng is an array, then argument endindx must be either a scalar integer or an array of the same length as argument rng.

EXAMPLES:

```
rngend(8:,15)   # returns 15
rngend(1:4, 3)  # returns 4
rngend([8:, 1:4, 7:], 3)
    # returns vector [3,4,3]
rngend([8:, 1:4, 7:], [11,12,13])
    # returns vector [11,4,13]
```

rnginc(rng), rnginc(rng, incindx) rng - the range (or array of ranges) whose stride (or strides) is to be returned.incindx - a value which is not used or checked. This argument need not be present.

This function returns the stride(s) of the given range(s) (argument rng). A second argument, incindx, is allowed but is not required (and is not used) in order to provide function RNGINC with an interface similar to RNGBEG and RNGEND. Defaulted values for range strides ar always 1. The first argument can be ny array in which case an array of increment fields is returned.

EXAMPLES:

```
rnginc(1:8,2)   # returns 1
rnginc(1:8) # returns 1
rnginc([1:8, 10:4:-1, 1:7:2])
    # returns vector [1,-1,2]
```

rngsetdf(rng, default_rng) rng - the range (or array of ranges) to be returned with any DEFAULTED fields replaced.default_rng - a range (or array of ranges) which has no DEFAULTED fields (increment fields are ignored). The corresponding fields will be returned in place of any DEFAULTED field in argument rng.

This function returns the given range(s) (argument rng) in which all the DEFAULTED fields have been replaced by the corresponding fields in the given default ranges (argument default_rng). If argument rng is an array, then argument default_rng must be either a scalar range or an array of the same length as argument rng.

EXAMPLES:

```
rngsetdf(:8, 2:10)  # returns 2:8
rngsetdf(2: , 1:22)
    # returns 2:22
rngsetdf(::3 , 1:15)
    # returns 1:15:3
rngsetdf([:8, 1:4,  7:], 2:15)
    # returns vector [2:8,1:4,7:15]
rngsetdf([:8, 1:4,  7:], [-1:5, 2:3, 3:9])
    # returns vector [-1:8,1:4,7:9]
```

rsum(x,idim) returns the partial sum of array x in reverse order. See “psum(x,idim)”. 21

shape(x), shape(x,n1,n2,...) shape(x), where x is an array, returns a vector that gives the shape of x (i.e., its ith component is the range of the ith subscript of x). All dimensions of length 1 are removed from x before the shape information is returned. If x is a scalar then shape(x) is 1. If the shape of x is one dimensional, shape(x) is a scalar giving the length.shape(x,n1,n2,...nk) returns x reshaped to the specified dimensions. It is an error if the length of x is not the product of those n1 through nk that are positive. If ni is negative, this is a so-called ”rubber index” signal. It indicates the number of repetitions of the data in the given dimension that are required. This is frequently used to create copies of data so as to match the shape of a higher-dimensional object with which it is used in arithmetic statements. The final number of elements in the result will be the product of the length of x with the absolute values of those ni that are negative.

EXAMPLES (reporting the shape):

```
integer x(3,2:4,1,5), y(3,2:4,2,5)
shape(x)    # returns [3,3,5]
shape(y(,,,1:4) # returns [3,3,2,4]
shape(0)    # returns 1
shape(iota(6))  # returns 6
```

EXAMPLES (changing the shape):

```
shape(x, 2, 3) # returns [[1,2], [3,4], [5,6]]
integer w = iota(3)
shape(w, -2, 3) # returns 2 by 3 matrix,
    # with each row = [1, 2, 3].
shape(w, 3, -2) \# returns 3 by 2 matrix,
    # rows = [1,1], [2,2], [3,3].
```

Suppose you have a variable x of shape (20, 35) and you wish to multiply each plane of y by it, where y is of shape (20, 35, 12). You would write: y * shape(x, 20, 35, -12)Note how you can read off the final shape of the result by taking absolute values.

sign(x,y) returns object x with the corresponding signs of the components of y attached to the components of x. If x and y are not the same size and shape, x must be a scalar, and it will be expanded into an object the same size and shape as y before the signs are attached, Both arguments must be integer or real, or double.

sin(x) returns the sine of object x, which must be in radians. See “cos(x)”. 21

sinh(x) returns the hyperbolic sine of object x. See “cos(x)”. 21

sngl(x) returns an object of the same size and shape as x whose components are those of x, converted to real.

sorti(&sortary, &sortidx, length) where sortary is an integer array to be sorted in ascending order and length is the number of integers to be sorted. The sorted list is returned in array sortary. Array sortidx is an output array of integers containing the permutation used to sort array sortary. The i-th value of sortidx is the index into the unsorted list of the i-th element in the returned sorted list.NOTE: an & is needed in front of both arguments sortary and sortidx since they return data.

spanl(start,stop,npoints) returns a list of floating point numbers logarithmically spaced, where start is the starting number, stop is the stopping number, and npoints is the number of points.

squeeze(x) array x is reshaped such that all dimensions of length 1 have been removed. If no such dimensions exist, then squeeze(x) is the same as x.

sqrt(x) returns the square root of object x. See “cos(x)”. 21

strchpat(s, oldpat, newpat) string substitute. Returns a string in which every occurrence of oldpat in s is substituted with newpat. if newpat is not specified, all occurrences of oldpat are removed.

strlen(s) returns the string length of string s.

struct accepts any number of arguments (including other structures) and returns a structure whose components are these objects. To access a component of a structure, one selects that component like an array component, by means of its subscript in parentheses.

substr(s,pos,len) returns a substring of s starting at the 1-origin index pos and is of length len.

sum(x,idim) returns the sum of array x. x can be of type integer, real, double, or complex. The resulting type is the same as x. If idim is supplied, then the function is applied to each group of elements in x whose indices vary only in the i-th dimension. The output array produced is the same shape as x, except that the i-th dimension is removed. If idim is not supplied, then the function is applied to the entire array, resulting in a scalar output.

sup can have an arbitrary number of arguments of any sizes and shapes. sup returns a scalar which is the maximum value present amongst all the components of all the objects. Arguments must be of arithmetic type; only the real parts of complex objects participate.

svd(x) svd(x)= singular value decomposition , structure (u, d, v) such that x = u *! diag(d) *! v', u, v unitary matrices, d vector of singular values.

tan(x) returns the tangent of the object x, which must be in radians. See "cos(x)". 21

tanh(x) returns the hyperbolic tangent of x. See "cos(x)". 21

tolower(s) converts a string from uppercase to lowercase.

toupper(s) converts a string from lowercase to uppercase.

transpose(x) transposes the matrix x.

trim(s) returns the string s with both leading and trailing blanks removed.

triml(s) returns the string s with leading blanks removed.

trimr(s) returns the string s with trailing blanks removed.

truerange(namex) where namex is string which is the name of an array or subscripted array. This function returns a matrix whose rows contain the lower and upper subscripts for each dimension of the array named by namex. If namex references a scalar range, returns `[1,1]`.The values returned by this function might be different than those returned by function rangex. Function truerange will return information about dimensions of length 1.

EXAMPLES:

```
integer x(3,2:4,1,5), y(3,2:4,2,5)
truerange("x")  # returns 4x2 matrix with rows
    #    [1 3], [2,4], [1,1], [1,5]
truerange("y(,,,1:4)")
    # returns 4x2 matrix with rows
    #    [1 3], [2,4], [1,2], [1,4]
```

trueshape(namex) where namex is a string which is the name of an array or subscripted array. This function returns a vector that gives the shape of the array named by namex (i.e., its ith component is the range of the ith subscript of x).The values returned by this function might be different than those returned by function shape. Function trueshape will return information about dimensions of length 1.

EXAMPLES:

```
integer x(3,2:4,1,5), y(3,2:4,2,5)
trueshape("x")  # returns [3,3,1,5]
trueshape("y(,,,1:4)")
```

type(x) returns the type of x as a string. Possible values are: "`integer`", "`real`", "`complex`", "`double`", "`logical`", "`character`", "`chameleon`", "`range`", "`function`", "`indirect`", "`structure`", "`word address`", and "`null`".

utype(t1,t2,...) Defines from one to ten user delimiters for use in COMMANDs. Each delimiter `ti` must be a string. Each call to utype creates a new table of user delimiters from scratch, and for subsequent reference in a COMMAND, they will be referred to by one of the digits 1, 2, ... , 9, 0 according to the order in which they occurred anmong the arguments of `utype`. `utype` is intended to be called as a subroutine, but if used as a function, it returns its last argument as its value.

vmax(x [,i)] with one argument, vmax(x) is the same as max(x), i. e., gives the maximum component of x, a scalar. vmax(x, i) performs the maximum over the ith dimension of x, delivering an array of one less dimension whose components are the maximum of x with those subscripts, as i varies. For example, if x(1,1) = 1, x(1,2) = 15, x(2,1) = 16, x(2,2) = 12, then vmax(x,1) is the vector [15,16] and vmax(x,2) is [16,15].

vmin(x [,i)] with one argument, vmin(x) is the same as min(x), i. e., gives the minimum component of x, a scalar. vmin(x, i) performs the minimum over the ith dimension of x, delivering an array of one less dimension whose components are the minimum of x with those subscripts, as i varies. For example, if x(1,1) = 1, x(1,2) = 15, x(2,1) = 11, x(2,2) = 12, then vmin(x,1) is the vector [1,11] and vmin(x,2) is [1,12].

where(cond,x,y) cond is an array or scalar of type logical. x and y each can be either a scalar or an array of the same length (but not necessarily the same shape) as cond; and their types can be either integer, real, double, or complex. Note: x and y do not have to be of the same type. If y is present, the output is the same size and shape as cond and its type is the more encompassing of x and y. For example, if x was of type integer and y was of type complex, then the output would be complex. All data will be coerced to the proper type before being stored into the output array. The value of the output array is calculated as follows. Those elements of the output array corresponding to true values of cond are set to the corresponding values of x (or set to x if x is a scalar). Those elements of the output array corresponding to a false value of cond are set to the corresponding values of y (or set to y if y is a scalar). If y is not present, then the output is a one dimensional array whose length is the number of true values in cond and whose type is the type of x. The values of the output array are those elements of x which correspond to a true value of cond. If x is a scalar, then all elements in the output array are set to x. Example: where([true,false,true],[1,2,3],0.0) evaluates to [1.0,0.0,3.0]

zcen(x) applies zone centering to the array specified.

CHAPTER

TWENTYTWO

User-Defined Functions

22.1 Defining Functions

The user can define functions to perform some task not available in the built-in functions. At compile time, user-defined functions are translated into intermediate code, which is not executed upon completion of the function definition, but instead is stored. Later the function can be invoked, like a built-in, by executing code that calls the function.

The skeleton used for function definition is as follows:

```
FUNCTION name formalparams
     <stlist>
ENDF
```

`name` is any user identifier of 128 or fewer characters; it must not be a keyword. If a user variable or function with the same name already exists it is deleted.

`formalparams` is an optional parenthesized list of identifiers separated by commas. These identifiers are interpreted in `<stlist>` as associated with the actual parameter values passed at run-time. These names may be chosen arbitrarily; when the function is invoked variables with those names come into being at the highest level of the search stack. Thus, if a user-defined variable exists, and a function is called with that same name as a formal parameter, the user-defined variable will be inaccessible while in that function. If `formalparams` begins with a semicolon, or contains a semicolon in place of one of the commas, the arguments that follow the semicolon are optional. The user may call such a function with none, some, or all of its optional arguments. See “Functions With Variable Numbers of Arguments” on page 188 to see what happens in that case.

The `<stlist>` in a function is unrestricted, except that functions cannot be nested. An attempt to define a function inside a function or other structured statement is not allowed and will result in an error.

There is no provision for declaring the formal parameters and the function name. The formal parameters act like chameleon variables at call time, in that they assume all the attributes of the associated actual parameters. They can be coerced to a particular type by the built-in functions int, float, or cmplx, if the user wishes.

22.2 RETURN

An object (and hence a chameleon-like type, size, and shape) is associated with the function name by the statement

```
RETURN <lexp>
```

which returns control to the calling statement with the object <lexp> as the "value" returned by the function. A simple

```
RETURN
```

returns no value. If flow of control in a function drops to the ENDF, then a RETURN is automatically executed.

22.3 Local Variables

User variables can be declared inside a function. If they are, these variables are dynamic, and are allocated space when the function is executed. They are then deallocated upon RETURN. These variables can have the same names as other user variables, and if so, they supplant those variables as long as the function is at the end of the call chain. Because these variables exist only during execution of a function, they cannot be accessed from outside that function, and hence are strictly local to it.

Prefixing the declaration of a variable with the keyword GLOBAL creates a variable which is visible inside all functions and which replaces any currently existing variable with the same name.

22.4 CALL Is By Value

Actual parameters are passed by value, meaning that at run-time, their values are computed and passed to the function to be linked to the formal parameters. Thus, assignment of a value to a formal parameter in a function will not alter the actual parameter in the calling routine.

If a function is to have side effects (i.e., change the value of some existing variable), then this may be done by accessing the variable globally: a variable is accessible to a function if it has been declared outside all functions (or is predefined), and if no formal parameters or local variables within the current function have that same name. Another method is to make the formal argument the name of the array, and use an INDIRECT variable to reference it (See "Indirect Variables" on page 67.)

A function can be invoked in either one of two ways. The first is exactly the same as for built-in functions: as an operand in an expression (function name followed by expressions for its actual

parameters in parentheses). The number of actual parameters must agree with the number of formal parameters (possibly zero) declared in the FUNCTION header.

The second way to invoke a function is to use the CALL statement

```
CALL name actualparams
```

where `name` is the function being invoked. `actualparams` is optional; it can be absent if name was declared with no parameters, and present (with the same number of parameters) when name was declared with parameters. Whether or not name returns a value is irrelevant to the CALL statement, since the value is discarded. Presumably, CALL only makes sense if used with a function that has side effects, say giving values to global variables, displaying values, or running physics packages.

If a function has no formal parameters then it can be called in any of the following ways:

```
CALL name
CALL name()
name()
name
```

but of these only `name()` can be used in an expression, and only `name` or `name()` return a value.

The meaning of a name is decided at execution time, so it is acceptable to define a function that calls a second function that has not yet been defined, as long as the second function is defined before the first function is executed. If this is not done, an error message will be received to the effect that the name does not exist. A function MAY call itself; logic to prevent infinite recursion is your responsibility.

22.5 Examples of User Functions

As an example of function definition, the following function computes the square root of its argument (if real and positive) within a specified tolerance:

```
FUNCTION myroot (x,eps)
REAL y= 1.,eps1 = max(abs(eps),1.e-12)
IF (x<0) THEN
     REMARK "myroot called with negative value";x
     RETURN 0.0
ENDIF
IF(x = 0) THEN RETURN 0.0
DO
     y = .5*(y + x/y)
UNTIL (abs(y*y-x) <eps1)
RETURN (y)
ENDF
```

The tolerance of the result is measured with eps. Note that 0.0 is returned if the actual argument was negative. In addition, an error comment is printed and x is displayed. If x is not negative, then either 0.0 is returned or else iteration proceeds until the desired tolerance is met.

The following function computes the value of n factorial. Note that this function calls itself recursively, which is allowed in Basis.

```
FUNCTION nfact (n)
IF (n<0) THEN
     REMARK "negative factorial not defined"
ELSEIF (n = 0)
     RETURN 1
ELSE
     RETURN n*nfact(n-1)
ENDIF
ENDF
```

CHAPTER

TWENTYTHREE

Compiled Functions

Certain packages, including the Basis parser, contain compiled functions and subroutines. These are modules written in Fortran or assembler which have been compiled and loaded into the executable program. Basis has the ability to execute some of these functions and subroutines in a way similar to the way Basis executes built-in and user-defined functions.

The difference between a function and a subroutine is that a function returns a value, while a subroutine does not. In what follows we will simply use the name “function” to mean “function or subroutine”.

In order for the function to be executable from Basis, the function must be listed amongst the variables of the package. That is, the author of the package had to incorporate a description of the function into his or her variable description file, which is part of the process of making a Basis code. The function will be listed with a “template” for its calling sequence, which will consist of a parenthesized list of arguments with optional types attached with a colon to the name, such as:

```
blah(x,y,z:integer,w:string) complex
```

This means that blah is a function that takes 4 arguments, the first two of type real (by default, since they have names that begin with letters other than i through n), the third one of type integer, and the fourth one of type “string” which means a character string of any length up to 500. The function blah returns a complex value.

The functions that are declared in the parser package are described in this manual. (43) For other packages, consult the documentation supplied by the author or poke around with the list command until you find some.

Compiled functions can be of type integer, real, double, complex, logical, or character*(n). Arguments to compiled functions can be of the same types. Currently, arguments to compiled functions cannot be the names of functions of any type. Compiled functions only return scalar values.

A compiled function that modifies one of its input arguments may be dangerous. Basis has no way of checking the length of an array expected by a compiled function, and a call to a compiled function that modifies a location not supplied in the call will typically cause Basis to crash. Also note that functions are called by value, and hence the modified argument is not accessible after the function returns (but see below). However, used properly, compiled functions can be a very powerful tool.

23.1 CALLing By Address

A (possibly subscripted) variable can be passed to a compiled function by address. (See “Indirect Variables”, 14.7 for the equivalent method for user-defined functions.) In this case modifications which the function makes to the array WILL change the original. To do this, precede the name with an ampersand in the calling sequence. For example, suppose zero(x,n) is a routine which sets the first n components of x to zero. Then

```
real x(10) = iota(10) ; call zero(&x,5); call zero(&x(7),1)
```

will result in x containing `[0.,0.,0.,0.,0.,6.,0.,8.,9.,10.]`.

Only compiled functions can be passed an argument by address. If the variable being passed by address is of any character type, it cannot be subscripted. When a variable is passed by address no type conversion is done.

CHAPTER

TWENTYFOUR

Defining Your Own Commands

24.1 The COMMAND Statement

```
fname COMMAND arglist
fname command_spec arglist
```

A formal description of this syntax will be developed step-by-step in this section. Informally, the COMMAND statement will cause function `fname` to be called with arguments `arglist`. This capability can be used in conjunction with the macro facility to define your own blank and/or comma delimited "commands". It is also possible to specify other delimiters between arguments, or even to define your own. In the following example

```
mdef mycommand = myfunction command mend
mycommand arg1, arg2
```

a command called `mycommand` is defined and is used. The above usage of this command (`mycommand arg1, arg2`) will cause function `myfunction` to be called with arguments `arg1` and `arg2`.

The formal definition of the syntax for argument specification is as follows. The first syntax form is a function name `fname` followed by the word `COMMAND` (or `command`) followed by a comma and/or blank delimited argument list – WITHOUT parentheses. This form of the COMMAND statement expects `arglist` to contain only expressions, i.e. all arguments are first evaluated. Thus all strings must be quoted.

The second form is identical to the first form except that instead of entering the word "`command`", you now enter a `command_spec` which is the word "`command_`" (note the underscore) immediately followed by a type specification. This specification is a series of `s`'s, `S`'s, `e`'s, and/or `E`'s, optionally followed by a parenthesized series of `s`'s, `S`'s, `e`'s, and/or `E`'s at the end. (This second form also allows you to specify what delimiters you wish as defaults between arguments, and any special delimiters between specified arguments. We will postpone the question of delimiters until a later section.) This specification allows you to have arguments which are either unquoted strings (`s, S`) or expressions (`e, E`), or a combination of both. The first letter of the specification defines the type of the first item in the COMMAND `arglist`, the second letter, the second item,

etc. The s denotes an unquoted string and an e an expression. The upper case letters S and E also denote unquoted strings and expressions, except that macro expansion will be suppressed; thus the macro-suppressing brackets "{}" are not necessary in the corresponding arguments. The letters of the specification within the ()'s (if any) are used repeatedly until the end of the command list (i.e. (se) is sesese...). If no ()'s are present in the specification, then the last letter is used repeatedly until the end of the COMMAND arglist (i.e. "se" is the same as "seee...").

The following example illustrates defining a new "command" called gotoit which expects a series of expressions as arguments. A second "command" called dothis is also defined, which expects a series of name and expression pairs.

```
define gotoit   f COMMAND
define dothis   g command_(se)
gotoit a,4+5,c
dothis x 6+1 y 9+3
```

The above example is equivalent to

```
call f(a,9,c)
call g("x", 7, "y", 12)
```

This is accomplished in two stages. First, "gotoit a,4+5,c" is expanded to "f COMMAND a,4+5,c". The COMMAND statement then evaluates expressions a, 4+5, and c and calls function f with the resulting values. In the next example, "dothis x 6+1 y 9+3" is expanded to "g command_(se) x 6+1 y 9+3". The "command_(se)" statement treats x and y as strings and evaluates expressions 6+1 and 9+3. Then function g is called with the resulting values. It should be noted that f and g can be any type of function such as a Basis function, built-in function, or compiled function or subroutine.

The above examples show that macros and COMMANDS can interact to provide a powerful tool. Besides being able to use a COMMAND in a macro, you can also use macros in a COMMAND arglist. If the arguments are specified by lower case e or s, then macros used in an argument are expanded, unless they are protected by curly brackets { }'s (or are enclosed in quotation marks). Macros are not expanded in arguments specified by upper case S and E, so it is not necessary to use the somewhat clumsy bracket notation to suppress macro evaluation.

Some examples follow which show two things – different ways to delimit a COMMAND arglist, and the usage of macros in an arglist.

```
define expr (6+2)
define x   str1
mdef y = str2 mend
mdef z = str3 mend
echo COMMAND a, (6+2)/2, c      # comma delimited
echo COMMAND a  expr/2  c       # blank delimited
echo COMMAND a expr/2,          # mixed delimited
```

```
               c                 #   continuation of arglist
echo COMMAND a {expr}/2 c        # one way to suppress macro
echo COMMAND_eEe a expr/2 c      # another way to do it
echo command_s str1, str2, str3 # comma delimited
echo command_s x y z             # blank delimited
echo command_s x y,              # mixed delimited
               z                 #   continuation of arglist
```

The first set of COMMAND statements is equivalent to `call echo(a,4,c)`, the second to `call echo(a,expr/2,c)` (one hopes that there is a variable named `expr` which is defined and has a value), and the third set to `call echo ("str1", "str2", "str3")`. Note that in the third and eighth COMMAND statements that both commas and blanks were used as delimiters in a single statement. A comma at the end of any COMMAND line signifies that the COMMAND `arglist` continues on the next line.

WARNING: It should be noted that

```
define expr 6+2
define exprnew  7 +1
mdef mystring = this is a string mend
echo COMMAND a, expr/2, c
echo COMMAND a, exprnew, c
echo COMMAND a, 7  +1,  c
echo command_s this is a string
echo command_s mystring
```

is equivalent to

```
call echo(a, 7, c)
call echo(a, 7, +1, c)
call echo(a, 7, +1, c)
call echo("this", "is", "a", "string")
call echo("this", "is", "a", "string")
```

The expression "`expr/2`" is equal to seven since this expression expands to "`6+2/2`". Macro writers should remember to enclose any expressions in parentheses, such as "`(expr)/2`", if they wish their expression to be evaluated before any other operations are performed.

The rest of the preceding examples deal with delimiting issues. Blanks and commas are the default delimiters between `command` arguments. Thus the blanks between the "`7`" and the "`+1`" and between the words in "`this is a string`" are considered delimiters, even if these blanks appear in the middle of a macro definition. If you have an expression with blanks that you wish to be considered one expression, then enclose the expression in parentheses ()'s. If you have an unquoted string with blanks (or commas) that you wish to be considered one string then enclose the entire string (or just the blanks or commas) in quotation marks (") or protection brackets { }'s. (The other method of specifying delimiters other than the defaults is discussed in a future subsection). Thus

```
define expr (6+2)
define exprnew (7 +1)
mdef mystring = {this is a string} mend
echo COMMAND a, expr/2, c
echo COMMAND a, exprnew, c
echo COMMAND a  (7  +  1) c
echo command_s "this is a string"
echo command_s mystring{ and here is some more}
```

is equivalent to

```
call echo(a, 4, c)
call echo(a, 8, c)
call echo(a, 8, c)
call echo("this is a string")
call echo("this is a string and here is some more")
```

24.2 Changing the Default Type of a COMMAND Argument

Suppose you define a new "`command`" `helpme` which expects a series of unquoted strings. Now what if someone using your "`command`" `helpme` wants to use an expression to calculate the value of a string? This user can override your default type specification of "`command`" `helpme` by typing a caret "^" (the user can have blanks following the caret) in front of the arguments whose type she wants to change. The caret will change strings to expressions and expressions to strings (it will not change the case, i. e., `s` becomes `e` and `S` becomes `E`).

EXAMPLES:

```
define helpme  g command_s
define exprs   f command
character*5 root = "mynam"; integer i=5
helpme x y z
helpme x ^ root//format(i,0) z
helpme x ^ (root // format(i,0)) z
exprs 1+2 3+4
exprs ^ abc ^ 3+4
```

The above examples are equivalent to

```
call g("x", "y", "z")
call g("x", "mynam5", "z")
call g("x", "mynam5", "z")
call f(3, 7)
call f("abc", "3+4")
```

Blanks are considered delimiters. The only difference between the second and third `helpme` commands in the above example is the spaces surrounding the // operator and the parentheses surrounding the expressions. Because of these blanks the parentheses are needed.

24.3 Specifying Other Delimiters in a COMMAND Statement

There is an additional COMMAND syntax which allows one to specify default delimiters (other than blanks and commas) for the entire command and to vary these delimiters between individual pairs of arguments; it also allows users to define their own delimiters and to specify those. Users can define their own delimiters by means of the new builtin function `utype`. One calls this function with a list of from one to ten strings, which are then entered into a table in order. For example

```
call utype("=","with","?","by")
```

will define user delimiter number 1 as `"="`, number 2 as `"with"`, and number 3 as `"?"`. There are no other user delimiters. (Each subsequent call to `utype` redefines the user delimiter list to whatever its arguments are.) When specifying user delimiters, the number of the delimiter (1 through 9, with 0 representing the tenth) is used.

Delimiters available from the system are blank and comma (which are always the defaults if nothing else is specified), the `at` symbol , and the equal sign `=`. These are denoted (respectively) by `W` (for "whitespace") `C`, `A`, and `Q`. The lower case letters `w`, `c`, `a`, and `q` are used to denote the suppression of a particular delimiter.

The syntax is that `command_` is followed by a string including one or more of the delimiter characters and 's'/'S', 'e'/'E' characters, with an optional set of parentheses, as follows:

```
command_<string1><string2>(<string3>)
```

where at least one of `<string1>`, `<string2>`, and `(<string3>)` must be present, and `<string2>` and `(string3>)` (if present) must begin with one of the argument designators `e, E, s,` or `S`, and:

<**string1**> (if present) is a string of delimiter characters and specifies the default delimiters between the arguments of this command. This may consist of up to four of the letters W/w (white space is/is not a default delimiter), C/c (comma is/is not a default delimiter), A/a ("at" is/is not a default delimiter), or Q/q ("equals" is/is not a default delimiter); and any combination of digits standing for user delimiters. The order is unimportant. If <string1> is absent it defaults to 'WC'.

<**string2**> (if present) consists of a list of the letters s (S) and/or e (E), each optionally followed by delimiter characters as enumerated above. Delimiter characters in between argument characters are used to modify the default delimiters between those two arguments only. They may be used either to specify additional delimiters or to enable (or suppress) one of the four standard ones.

(<string3>) (if present) consists of a parenthesized list of argument and delimiter designators as in <string2>. The parentheses mean to repeat <string3> as often as necessary to include the rest of the arguments. If (<string3>) is not present, then the last argument designator in <string2> will be applied repeatedly, if necessary, to cover all specified arguments. If <string2> is absent as well, then all arguments will default to expressions with macro expansion enabled (e). Note that (<string3>), if present, is required to contain at least one argument designator, and it must be the first character after the "(".

Here are some examples. We shall assume in what follows that utype has been called to set up user delimiters "?", "with", "%", "by", in that order.

```
f command_wse the first argument, ( 6 + 8) * 43 ,88
```

is equivalent to

```
call f(" the first argument",(6+8)*43,88)
```

Note that the initial w suppresses the use of white space as delimiter, so the only default delimiter left is the comma. White space is gathered as part of the string argument, but has no significance in the expression arguments.

```
define name sam
define macpak specialvar
g command_s2SQe1(e1) name with macpak = 4 ? 3 ? 2 ? 1.56
```

is equivalent to

```
call g("sam","macpak",4,3,2,1.56)
```

Here the first string, name, is expanded as a macro, while the second, macpak, is not, because it was specified with a capital S. The number 2 user delimiter with was used between the first two arguments, then = as specified by Q, and then the rest of the delimiters are number 1 (?) as specified by the repeated designation e1 in parentheses. Although white space is a default delimiter throughout this command, note that white space which surrounds non-blank delimiters is ignored. Indeed, white space is necessary surrounding with because it would not be recognized as a separate token otherwise.

```
h COMMAND_SQe2e4e name = 6 with 18 by 2
```

is equivalent to

```
call h("name",6,18,2)
```

This time name is not expanded because it was specified by upper case S. Equal (Q) and user delimiters with (2) and by (4) were specified as additional delimiters between the second and third, and third and fourth, arguments, respectively. The occurrence of 2 and 4 only ALLOWS the with and by to be used as delimiters. Comma and white space are still valid delimiters unless specifically suppressed by lower case w and c.

24.4 No Delimiters at All: the COMMAND_L

The COMMAND_L is a special construct not previously mentioned. It allows the entire line following the 'command_l' to be gathered as one argument; nothing is accepted in any way as special except the end of the line. Thus in effect 'command_l' grafts quotes onto the beginning and end of the rest of the line. For example,

```
parsestr command_l integer x = shape(iota(27),9,3); x #define x
```

is the same as

```
call parsestr(" integer x = shape(iota(27),9,3); x #define x")
```

whereas

```
parsestr command_wcs integer x = shape(iota(27),9,3); x #define x
```

is equivalent to

```
call parsestr(" integer x = shape(iota(27),9,3)"); x #define x
```

The action is quite different; in the first case, the entire rest of the line is picked up and enclosed in quotes; in the second case, argument gathering stops with the semicolon. (This is intended to illustrate that either a semicolon or a pound sign will normally cause the gathering of COMMAND arguments to cease. COMMAND_L is intended to be a way of getting around this.) In the second case, `x` may be an unknown variable, globally, or if known, will almost certainly be different from the array defined locally by `parsestr` and then lost upon return. To illustrate this, here is a sample Basis run:

```
Basis> character*4 x = "abcd"
Basis\> parsestr command_l integer x = shape(iota(27),9,3); x
x                   shape: (9,3)
 row col =       1  2  3
 1:     -        1 10 19
 2:     -        2 11 20
 3:     -        3 12 21
 4:     -        4 13 22
 5:     -        5 14 23
 6:     -        6 15 24
 7:     -        7 16 25
 8:     -        8 17 26
 9:     -        9 18 27
Basis> x
x        = "abcd"
Basis\> parsestr command_wcs integer x = shape(iota(27),9,3); x
x        = "abcd"
```

Notice how neither command destroys the global value of `x`; but the first command prints out the local value (thus showing that its entire argument has been enclosed in quotes), while the second prints out the global one (showing that the semicolon has been recognized as a statement separator, and hence as the end of the command argument).

With some care, the `l` argument specifier may be used in combination with other arguments, observing the following constraints: `l` (or `L`, which has exactly the same meaning) must be the last letter present in <`string2`>, and (<`string3`>) can not be present. This ought to be obvious after a moment's thought, because if `l/L` suppresses the recognition of all delimiters, then there is no way to collect an argument following the one it specifies.

For example,

```
f command_eel 6+8*4 5-3*7+2 string ; x # argument
```

is equivalent to

```
call f(6+8*4,5-3*7+2,"string ; x # argument")
```

CHAPTER

TWENTYFIVE

The Search Stack

Basis designates one package as the “current” package; this allows the user to specify one package to be searched first during display of values or listing of variables. The parser itself is a package named “par”. Basis begins with “par” as the current package, but the user can designate a new current package by entering PACKAGE pkgname. If desired, the user can specify several packages in order to create a “search stack”. The commands POP and PACKAGE pkgname can be used to manipulate the search stack. The current package is the package at the top of the stack. “par” is always present at the bottom.

The search stack is initialized by the program author. The command `list packages` can be used to see the current search stack. Consult the documentation for your particular program.

In searching for a variable or function name, Basis searches in the following order: first, the local variables and formal parameters of the currently executing function; second, the user-defined variables and functions; next, the packages on the search stack are examined in order. This stack always ends with the variables of the parser itself, the package called “par”. The searching is done at execution time, not compile time.

CHAPTER

TWENTYSIX

Package Control Statements

```
package pkgname
```

places `pkgname` at the top of the search stack, making it the current package. If `pkgname` was already in the search stack, it is moved to the top.

The routine `parpop` removes the top element of the search stack, making the next element the current package. If the stack has been reduced to `par` there is no effect.

CHAPTER

TWENTYSEVEN

The CTL Package

Some Basis programs use a special package named `ctl` to control the execution of physics packages. You can check if `ctl` is present in a program with the command `list packages`. If it is, commands named `run`, `generate`, `step`, and `finish` will be defined for you. These commands are in chapter 76 in *The Basis Package Library* document.

CHAPTER

TWENTYEIGHT

Removing Functions and Variables

Users may sometimes want to delete one or more user-defined functions or variables that are no longer needed. The FORGET statement allows this to be done. A simple

```
FORGET
```

wipes out all user-defined functions and variables, and releases the space occupied by them so that it can be reused.

```
FORGET name1, name2, ...
```

where name1, name2, are the names of a user-defined functions or variables, deletes those names and releases the corresponding space.

If the user want to delete a macro, he should use the UNDEFINE command.

If the user wants to protect user-defined functions and variables made up to this point from future FORGETs he can enter:

```
call protect
```

CHAPTER

TWENTYNINE

LIST Command

One important feature of Basis is that it knows about the variables in the different packages and can tell the user about them. The author of a package organizes the variables into "groups". The command LIST is used to display information about variables, groups, and packages.

```
LIST [ name ]
```

where name is the name of a variable, group, or one of the keywords packages, macros, groups, variables, or functions.

LIST LIST (with no argument) displays a help package for the LIST command.

LIST macros LIST macros displays a list of the macros that have been defined.

LIST packages LIST packages displays a list of the packages in Basis, giving the name of the package, a short description, and its current status.

LIST pkg.variables displays a list of the variables (and functions) in that have been declared in package `pkg`, sorted by group. If `pkg` is omitted it defaults to the user-defined variables.

LIST pkg.groups LIST groups displays a list of the groups in the package `pkg` with a short description. A group is a group of variables that the author of a package has designated as logically related to one another. If `pkg` is omitted it defaults to the user-defined variables.

LIST pkg.functions LIST FUNCTIONS displays a list of functions in the package `pkg`. If `pkg` is omitted it defaults to the user-defined functions.

LIST Group LIST Group displays information about all the variables in the group named Group. Group must be entered with correct case; at least the first character will be upper case. The name of the group may be abbreviated to any unique prefix. One special group is the group User, which contains all the variables and functions declared by the user. When a user function is executing, there is a special group named `Locals_fname` where fname is the name of the function. This group can be edited or listed while the function is executing or while any function called by it is executing. If fname calls itself, only the most recent incarnation can be viewed in this way.The parser package par contains two groups `Builtin_Functions`

and `Compiled_Functions`. LIST Builtin displays a list of the built-in functions such as sqrt. (See Chapter 21, “Built-in Functions” for a full description of the available functions.) In general, all the Fortran intrinsics are available, in generic form. Thus for example, one can use sqrt(x) to get the appropriate square root of x whether x is integer, real, double, or complex. LIST Compiled displays a list of compiled parser functions.

LIST name LIST name displays information about the name, including type, length, location, whether or not it is dynamic, its dimension, and a comment about it made by the package writer, and its attributes. If name is the name of a function, some different information about it is displayed. If name is the name of a macro then the definition of the macro is displayed along with whether it was declared with or without arguments.The user may prefix the name by a package name and a period, e.g., vf.sigcoef where vf is the package name and sigcoef is a variable name.Variables local to a function are visible ONLY when the flow of execution is currently in that function; such variables are NOT visible in functions that are called by it. There is a way to see such variables, however, for debugging purposes: include the statement `Locals_fname` in the function.

CHAPTER

THIRTY

Obtaining and Setting Scalar Values

The following functions accept an argument of type character, which should contain the name of a variable; if the variable is a scalar of the right type, the function returns its value. If it is the wrong type, or is not a scalar, or does not exist, then `kaboom` is called. Otherwise, these functions can be used in any expression.

```
ibasis(s: string) integer function
--return an integer value

rbasis(s: string) real function
--return a real value

dbasis(s: string) double function
--return a double precision value

cbasis(s: string) complex function
--return a complex value

lbasis(s: string) logical function
--return a logical value

sbasis(s: string) character*MAXSTRING function
--return a string value.
```

Since the value returned by `sbasis` is `MAXSTRING` characters long, there may be lots of extraneous blanks on the end. To get rid of these, you could define a macro, which would trim trailing blanks before returning the value. Here's an example without the macro, using `sbasis` just as written:

```
Basis> character *20 s = "Short String"
Basis> sbasis("s")
sbasis   = "Short String"
Basis>
```

Note the long string of unnecessary and distracting blanks. On the other hand, with the following macro definition,

```
mdef sbasis() =  trim({sbasis}($1))  mend
```

the following exchange would result:

```
Basis> character *20 s = "Short String"
Basis> sbasis("s")
trim     = "Short String"
Basis>
```

These next subroutines are complementary to `ibasis`, `rbasis`, etc., described above. They accept a string and a value of the appropriate type; if the string contains the name of a scalar variable and the type is right, then the subroutine assigns the value to the variable. The routines are:

```
sibasis(s: string, v: integer) subroutine
--set an integer value

srbasis(s:string, v: real) subroutine
--set a real value

sdbasis(s: string, v: double) subroutine
--set a double precision value

scbasis(s: string, v: complex) subroutine
--set a complex value

slbasis(s: string, v: logical) subroutine
--set a logical value

ssbasis(s: string, v: string) subroutine
--set a string value.
```

In the case of `ssbasis` the assignment follows the usual FORTRAN rules: if the source string is too long for the destination, it will be truncated from the right. If it is shorter than the destination, then the destination will be blank-filled on the right.

CHAPTER

THIRTYONE

Help and News

The function `news` displays information about recent changes. The files news and newslog inside `basis` contain the recent and cumulative news respectively. To invoke news, just enter `news` at the prompt.

The function `help` displays information about how to get further help on Basis and/or the physics package you are using.

Both `help` and `news` are simply compiled functions being executed by the Basis interpreter; these no longer are keywords.

CHAPTER

THIRTYTWO

Input, Output, and External File Access

32.1 Reading Basis Code From a Text File

```
READ filename
```

shifts input from the current source to filename, a text file created in advance by the user that contains Basis statements. After all statements in filename have been read, parsed, and executed, input resumes from the current source, including the rest of the line on which the READ command occurred. READ commands can also occur in files, to a depth of up to ten files. As the commands are read, they are displayed on the terminal unless an

```
echo = no
```

or

```
echo = logonly
```

command has been given. It may be desirable to put comments in the input files; this can be done by prefixing them with a pound sign (#). Everything on a line after a pound sign is taken as a comment. If `echo = no`, the user can still use the REMARK "message" command to display progress reports on the terminal.

If filename is not in the current working directory, then Basis looks for filename in the following default directories in the order specified:

1. The directory from which the Basis source is being executed.
2. The directory specified by the environment variable WRK (if defined).
3. The directory specified by the environment variable HOME (if defined).
4. The directory $BASIS_ROOT/include (if environment variable BASIS_ROOT is defined.)

The Basis functions pathadd and pathrm may be used at runtime to add or remove other directories to and from the search path. pathadd (”directory-name”) adds ”directory-name” to the top of the list and pathrm (”directory-name”) removes ”directory-name” from the list.

In addition, the Configure file has keywords codefile and path, which allow the user to specify at compile time additional search paths. The reader is referred to *Writing Basis Programs: A Manual For Authors*, page 404, for a discussion of these keywords.

Basis displays an error message if it fails to find the specified file.

IMPORTANT: The READ command should not normally be used in combination with Basis compound statements such as DO loops, IF-THEN statements, FUNCTION’s, and so on. A READ in the middle of a compound statement is not executed until the compound statement has finished.[1] If more than one READ occurs inside a structured statement, they will be executed after that statement completes, but in the reverse order of their occurrence. For example, the following code sequence:

```
if (x == 3) then
   read a3
   read a4
   << "I just finished reading a3 and a4."
endif
```

is executed in the order as if it were written:

```
if (x == 3) then
   << "I just finished reading a3 and a4."
   read a4
   read a3
endif
```

Thus, while it is not illegal to put READ statements inside compound Basis statements, it is almost never correct.

If you wish to skip the first n records of the file, enter:

```
nskipr = n
```

before your READ command. Basis will automatically reset `nskipr` to 0.

[1] More precisely, *execution* of a READ statement simply opens the file and pushes its descriptor onto the stack of files from which Basis will read next. Processing new text from the next file does not actually begin until the (largest) enclosing, compound, statement has finished.

32.2 Resuming Reading

```
RESUME [n] [filename]
```

RESUME resumes reading a file after a crash has occurred while reading some file. If filename is entered Basis resumes reading from filename, otherwise it resumes in the file where the crash occurred. If n is entered reading starts at line n, rather than with the line where the crash occurred.

32.3 Printing Messages on the Terminal

```
REMARK "message"
```

displays message on the terminal. This can document progress in executing the file when the variable echo equals no. If v is a character variable or expression, REMARK v prints the contents of v to the terminal.

32.4 Changing the Destination of Basis Output

```
OUTPUT TO filename
```

causes most subsequent output to go to the file filename. OUTPUT TO TTY closes the output file and returns Basis to writing output on the terminal. OUTPUT TO GRAPHICS sends the output to the plot file.

CHAPTER

THIRTYTHREE

The Stream I/O Facility

33.1 Introduction to Stream I/O

Stream input and output features are available in Basis. The user may read input from an existing file. The user may create an output file, or send output to a plot or terminal. This section discusses how each of these tasks may be accomplished. First we show how to use the function `basopen` to open an input file or create an output file. Then we introduce the Basis input operator >> and show how it can be used to read data from an input file. Next we introduce the Basis output operator <<, illustrating how to send output to an output file, to a terminal or to a plot. We then discuss how the user can format output by using the function format. Lastly, we show how to use the subroutine `basclose` to close files that have been opened using `basopen`.

33.2 Opening and Creating Files

To read input from an existing file or create an output file, the user must first open the file by calling the function `basopen`. Any attempt to read input from a file or send output to a file without first using `basopen` to open that file will result in an a semantic error. It is not necessary to call `basopen` when sending output to a terminal or plot.

The function `basopen` is an integer function that accepts two arguments: a filename and a specification. It returns a unit specifier which is used to direct output to and retrieve input from a specific file.

The general form of the function call is:

```
unit = basopen(filename, filespec)
```

where:

unit is a unique unit specifier whose value is set by `basopen`. The unit specifier is used in the input and output commands to specify the file being used. Once the unit specifier has been assigned a value by `basopen`, it should not be altered.

filename is the name of the file being opened. Its length can be up to 128 characters.

filespec filespec should be ”r” or ”w” or ”i”.A sequential formatted file is created, opened or inquired about. `kaboom` is called if anything goes wrong.When rw = ”r”: `basopen` searches for the file in the current directory and then in any lib libraries specified in the variable path (or in directories on Unix systems).If a file must be found in a lib library on NLTSS a copy is made to a temporary file with a different name. This file is destroyed when the program terminates.No copies are made when opening files in UNICOS or SUNOS directories.When rw = ”w”: if an error occurs, this file will be closed.When rw = ”i” the file is not opened; instead, OK or ERR is returned, indicating whether or not filename could be opened for reading.

It is possible to have several files open at once, provided each file has its own unit specifier. This means the user should use a different integer each time he or she opens a new file. Here are a few examples of how to create output files and open existing input files:

Creating output files:

```
outunit = basopen("newfile", "w")
number  = basopen("one", "write")
junk    = basopen("asdf","WRITE")
mine    = basopen("myfile",  "W")
```

Opening input files:

```
infile = basopen("mydata",  "r")
myin   = basopen("data1","read")
x      = basopen("input",   "R")
in     = basopen("test", "READ")
```

Note that abbreviations may be used for `"read"` and `"write"`. The two most important things to remember are that the unit specifier is an integer and must not be modified by the user once it has been set by Basis in `basopen`.

`basclose(unit)` should be used to close files opened with `basopen`.

33.3 The Input Operator >>

Basis stream input must be read from an existing file or from the terminal. If input is from a file, then the file must have already been opened using the function `basopen` (as described above) before any input can be read. Once this has been done, the input command may be used to retrieve input from the open file.

As a general rule, Basis stream input can read files created by Basis stream output (see See “The Output Operator <<” on page **??**.) There is one important exception to this rule. Double precision

numbers with three-digit exponents are sent out without a “D” preceding the exponent, regardless of formatting (FORTRAN does the same thing). The stream input lexical analyzer will therefore see the double precision number as a real (the mantissa) followed by an integer (the exponent). Although this may seem at first glance undesirable, this is precisely the behavior of FORTRAN on unformatted input.

Stream input from files may be done in two modes, noisy and non-noisy. Non-noisy mode might be used to read real, integer, and double precision numbers (complex numbers are not presently supported) from a text file which was produced as output by a FORTRAN program. This file may contain the numbers in tabular form and might include explanatory text and other non-numeric information. In non-noisy mode, all non-numeric items in the file are considered to be “noise” and are ignored. In noisy mode, the so-called “noise” is not ignored, but will be read in as character strings. WARNING: character string stream I/O will only work if the strings contain no imbedded blank characters except for trailing blanks. Even trailing blanks can’t appear if character arrays are being processed. Noisy mode will be explained in more detail below.

The Basis input statement consists of a unit specifier and one or more input variables or arrays. The general form of this command is:

```
unit >> var1 >> array1 >> var2 >> var3 ...
```

where

`unit` indicates the file from which the input is coming. It is the unit specifier which was returned by `basopen` when the file was opened. If the unit specifier is omitted, the terminal is assumed, and an input prompt will appear there.

`var1, etc.` indicates the variables and arrays to which the input values are being assigned. The user may input array elements or entire arrays.

Exactly how the input is assigned to the variables depends on the setting of the built-in variable “`noisy`” (whose default value is “`no`.”) This value may be set to “`yes`” or “`no`” by assignment, thus:

```
noisy = yes
```

When “noisy” is “no,” then numerical tokens (i. e., legal FORTRAN integers , reals, and doubles) are extracted from the input in the order that they occur and are assigned to the variables (“var1”, “var2”, etc.) also in the order of occurrence. All other characters in the input, which are either delimiters (currently, spaces and commas) or are not FORTRAN integers, reals, or doubles are treated as “noise” and ignored. The input command extracts the next available numbers from the input file, even if it must go to subsequent lines of the input file to do so.

In this mode all input variables and arrays must be numeric (integer, real, double). If an attempt is made to read to a non-numeric variable or array, then an error diagnostic occurs and the input file is closed (assuming it is not the terminal). If an attempt is made to read past an end-of-file, then

the built-in variable "`eof`" is set to "`yes`", but the unit is not closed unless a second attempt is made. (see "Detecting End-of-File" on page 148 for more details.)

The following is an example of "non-noisy" operation (so-called because "noise" is ignored, i. e., filtered out):

```
i1 = basopen("test","read")
integer i
real x, d(2,2)
i1 >> i >> x >> d
```

Let us suppose that input file "`test`" consists of the three lines:

```
c special input file
first = 2.56 , second = 13.51e-2
d = 1.2 2.3 3.4 4.5
```

Then after the execution of the above sequence of instructions, i will be 2 (the real value 2.56 having been coerced to integer), x will be .1351, d(1,1) will = 1.2, d(2,1) = 2.3, d(1,2) = 3.4, and d(2,2) = 4.5. The remaining characters in the file (the "noise") will have been ignored.

To understand noisy mode operation (`noisy = yes`, i. e., "noise" is no longer ignored), it is first necessary to understand how the stream input *parser* interprets the incoming text. The parser divides the input stream into what we shall call "tokens", based upon the principle that it will build the longest legal token possible at each step. These tokens are as follows:

names begin with '%', '$', or a lower-case letter and may consist of zero or more additional '%', '$', '_', digits, or letters of either case.

group names begin with a capital letter and may consist of zero or more additional '%', '$', '_', digits, or letters of either case.

integers an optional sign followed by one or more contiguous digits.

reals an optional sign followed by one or more contiguous digits either containing a decimal point, or followed by 'e' or 'E' followed by an integer, or both.

doubles an optional sign followed by one or more contiguous digits either containing a decimal point, or followed by 'd' or 'D' followed by an integer, or both. Note that a double with a three digit exponent that has been written out by the stream output operator will not contain the 'd' or 'D' in its representation, so that only the first part of the number will be accepted.

strings contiguous non-delimiters which are not one of the previous five types of token.

Tokens are separated from one another by delimiters (currently spaces and commas). However, sometimes delimiters are not needed to separate tokens; e. g., "123abc" will be recognized as

"123" followed by "abc"; "abc.123" will be split into "abc" and ".123"; but "abc123" is a single token. For a more complicated example, "Joseph_Q._Jones" is three tokens, namely "Joseph_Q," "._," and "Jones."

In non-noisy mode, as noted previously, all tokens are ignored except for integers, reals, and doubles. In noisy mode, however, all tokens are significant, and there must be a variable in the input stream corresponding to each token. Furthermore, those variables corresponding to non-numeric tokens must be of character type or else chameleons. Use built-in function `type` to determine what has been read into the chameleon.

Consider, for example, the file "test" used in the preceding example. The following code will give i the value 2 and x the value .1351, as before, but will in addition assign to `name1` the string "`first`" and to `name2` the string "`second`":

```
i1 = basopen("test", "read")
integer i
real x
character*12 name1, name2
i1 >> $a >> $a >> $a >> $a #skip tokens on first line
i1 >> name1 >> $a >> i >> name2 >> $a >> x
```

There are a number of important points to note from this example:

1. `$a`, a chameleon, is used as a "sink" to receive unwanted portions of the input. Each unwanted token must be read to `$a`; it takes four such assignments, for instance, to discard the first line.
2. Note that blanks and commas *separate* tokens but are not themselves tokens. Thus the "... `i >> name2`..." in the second line of stream input automatically reads over the blanks and comma separating "2.56" from "second."
3. The value of `$a` after all of this is "=".
4. The first line of stream input could be replaced by i1 >> return
5. See section "Skipping Input Data" on page 149 for details.

We will now show three equivalent ways of retrieving input from a sample file called "data". "data" is a very short file consisting of six integer values, as shown below. Assume that i, j, k, l, m, n, and o are integers. "data" looks like this:

```
21 453  1
56,34   98765454
```

Method 1:

```
i=basopen("data", "read")
i >> j >> k >> l >> m >> n >> o
```

Method 2:

```
i=basopen("data", "read")
i >> j >> k >> l
i >> m >> n >> o
```

Method 3:

```
i=basopen("data", "read")
i >> j >> k
i >> l >> m
i >> n >> o
```

Each of the above methods opens the file “data” and reads the contents of “data” into the variables j, k, l, m, n, and o.

The user is allowed to have more than one input file open at once, up to a maximum (currently) of five, not including the terminal. If Basis is in the middle of an input line in one file when the user asks it to read input from a different file, it will keep its place in the unfinished line and resume from there if subsequently requested again to read from that file. If an error occurs in the read (either because of an incorrect assignment such as number to character, or because of an input error), then all open files will be closed.

33.3.1 Detecting End-of-File

It is the user’s responsibility to determine whether the end of a file has been reached. For this reason an end-of-file flag (`eof`) has been provided. `eof` is an integer which contains the value `no` if the last read attempt was successful, and `yes` if the last read attempt was unsuccessful. The user should test if `eof` is `yes` when performing input, so as not to attempt to read past the end of a file.

When the end of a file is encountered, the variables that cannot be assigned new values because of lack of input retain their original values. Once `eof` is `yes` for a specific file, the user should make no further attempt to read input from that file.

Suppose, for example, we have an input file called “data”. Assume there are a variable number of inputs on each line, and an unknown number of lines. Once again i and j are integers. The user may read the input file as follows:

```
integer i,j
i=basopen("data","read")
i >> j                  # read first value
```

```
while (eof <> yes)       # if last read was successful
     call dostuff(j)     # process the value
     i >> j              # get next value
endwhile
```

Remember that `eof` indicates whether the last read was successful, and if the last read was not successful, j will retain its last value. Note also that `eof` is set by *any* unsuccessful read; if the read failed because of some kind of error, then the file will be closed. However, it will still be open if an actual `eof` was detected, and it is the user's responsibility in this case to detect the `eof` and close the file.

A couple of further words to the wise are in order. If `eof` becomes `yes` during a read operation involving several variables, even in the middle of a loop or if there are further variables to be read before `eof` is next tested, no error will result, and the subsequent variables simply will not be read. Finally, there is only one `eof` variable. If you happen to be doing alternate input from two or more different files, then `eof` could be set to `yes` by one file, and then reset to `no` by reading from the next. Thus one must be careful to test for `eof` before switching files.

33.3.2 Skipping Input Data

Basis provides a mechanism that allows users to skip certain portions of an input file. The word "return", used in an input command, tells Basis to ignore the remainder of the current input line, and to retrieve the next input from the following line.

As an example, consider the input file "junk" shown below. It is a file of integers:

```
23          45   56
98          76   54
12          34   78
89          21   43
67          90   87
```

Suppose the user only wants to read the first inputs on the second, fourth and fifth lines. This could be done as follows:

```
integer i,j,k,l
i=basopen("junk","read")
i >> return >> j >> return >> return
i >> k >> return >> l
```

The use of "return" depends upon where the parser is in the input line, and on the contents of the unread portion of the line. If there are non-null tokens yet to be read from the line, then a "return" causes parsing to skip to the start of the very next line. However, if the parser has fetched the last token in the line, there may be no characters at all left in the line, or the line might still

have characters on it which are only delimiters (and thus, possibly, invisible). In either case the "return" causes the parser to skip the next line and resume at the beginning of the second line following. This feature makes it unnecessary for the user to have to know whether input lines are blank-terminated before deciding how many "returns" to use to skip subsequent lines. Consider the following code (applied to the same file "junk"):

```
integer u,i,j,k,l
u = basopen("junk","read")
u >> i >> j >> k
u >> return >> l
```

After this sequence of instructions, i, j, k, and l will have the values 23, 45, 56, and 12, respectively. The "return" caused the second line to be skipped, even though the parser may still have been positioned before the end of the first line (because of the presence of blanks at the end of the line).

A user reading input from two or more files can use "return" as above to position the parser in the files. Basis always remembers its position in each of the opened stream input files. Thus interleaved "reads" and "returns" addressed to different files will always work properly.

33.4 The Output Operator <<

The user may direct Basis output to a terminal, to a plot, or to a file. For output to the terminal or a plot, invoke the output command as described below. For output to a file, first open the file using the function `basopen`, as described above.

We will now discuss the default Basis output command. The form of this command differs slightly depending on whether the output is being sent to a terminal, to a plot, or to an output file. The three forms of the Basis output command are:

```
<< output1 << output2...           #  output to a terminal
plot << output1 << output2...      #  output to a plot
unit << output1 << output2...      #  output to a file
```

where:

`output1, etc.` are the outputs. These may be integers, reals, doubles, or character strings. They can be scalars or arrays. Character arrays will presently only work if they do not contain any imbedded blanks.

`unit` is the unit specifier of the file to which the output is being sent (the result of the call to function `basopen`).

By default, each use of the output command produces one or more lines of output. If there is more output specified in the output command than will fit on one line, Basis will continue the

output onto extra lines. The exceptions to this are single strings that are longer than the maximum output line length of 80, and output commands using carriage control (see section on CARRIAGE CONTROL, below). If a string is longer than 80 characters, the first 80 characters of the string will be sent to the output unit. The remainder is discarded.

No spacing between outputs is provided by Basis. It must either be done explicitly or by use of the function format (see ”The Format Function” on page 152.) Here are some examples of `<<` output:

```
<< "This sends output to a terminal."
plot   << "Or to a plot."
x      << "Or even a file that has been opened."
ounit  << "i=" << i << "  " << "r=" << r
```

33.4.1 Carriage Control

Basis automatically provides a carriage return for each output command. Additional carriage returns may be inserted by the use of the word return in the output command. `return` may appear anywhere in the output command, and may appear as many times as the user wishes. For example:

```
<< return
<< x << return << y << return
```

Note that when `return` appears as the final output, the result is actually two carriage returns since one is still supplied automatically by Basis. A switch is available to suppress the automatic carriage return of the output command. By default, `autocr` is set to yes. The user may stop the automatic carriage return by setting `autocr` to no. Output is then buffered until a return is specified or the line buffer is exceeded, at which time the line is output. To stop the suppression of the carriage return, reset `autocr` to yes. For example:

```
autocr = no
i=4
<< "i=" << i << return
<< "j="
j=i+1
<< j << return
autocr = yes
<< "DONE"
```

produces the following output:

```
i=4
j=5
DONE
```

The user must exercise caution when output is being sent to more than one unit and the automatic carriage return is off. If the buffer is not empty when output is sent to a different unit, the buffered output may be sent to the wrong unit. Using RETURN at the end of an output command before sending output to a different unit will ensure that the buffer is cleared.

33.5 The Format Function

The user can format Basis output by using the function `format` which converts a numerical value to a character string. This string can then be used as an output in output commands, plot labels, etc. `format` needs two or four arguments depending on what type of number is being converted. Variations of the format function call are illustrated by this statement:

```
<< "iquad = " << format(iquad,0) << ", pi = " << \
  format(pi,0,5,1) << ", deficit > " << format(2e11,0,1,0)
```

which prints `iquad = -1234, pi = 3.14159, deficit > 2.0e+11`

33.5.1 Formatting Integers

`format` requires two arguments to convert an integer to a string variable. It needs the integer being converted, and the length (or field width) of the resultant string.

The general form is:

```
str = format(ival, fw)
```

where:

`str` is the character string returned by format. Ival is right-justified within str and str is blank-filled to the left.

`ival` is the integer being converted to a string.

`fw` controls the field width. If > 0, fw is the length of str. If fw = 0, str is just the length needed, without blanks. The field width must also be of type integer, and must be not be greater than the maximum length of an output line (132).

The maximum number of digits which may be converted using format is 14. If the user attempts to convert more than 14 digits, the resultant string will have an “r” in the right-most position. Likewise, Basis places an asterisk (`*`) in the right-most position if the field width is specified to be too small to hold the value being converted. Here are a few examples of correct and incorrect calls to format when the user wishes to convert an integer to a string. The resultant strings are also shown.

CALL TO format	RESULTANT STRING
str=format(784,0)	’784’
str=format(-456,0)	’-456’
str=format(784,6)	’ 784’
str=format(4.3,5)	ERROR: first argument is real
str=format(6,7.2)	ERROR: field width is real
str=format(78,567)	ERROR: field width too large
str=format(786,2)	`' *'` – field width too small
str=format(-456,3)	`' *'` – field width too small
str=format(9876543210987654,20)	`' r'` – too many digits

33.5.2 Formatting Reals and Doubles

`format` requires four arguments to convert a real value to a string. The user must provide the real number being converted, the length (or field width) of the resultant string, the number of digits to appear after the decimal point, and the form of the resultant string.

The general calling form is:

```
str = format(rval, fw, nd, ts)
```

where:

`str` is the character string returned by format. rval is right-justified within str, and str is blank-filled to the left.

`rval` is the real number being converted to a string.

`fw` controls the field width. If > 0, fw is the length of str. If fw = 0, str is just the length needed, without blanks. The field width must also be of type integer, and must be not be greater than the maximum length of an output line (132).

`nd` is the number of decimal places desired in str.

`ts` is the specification of the format of str. ts may be 0 to indicate D or E-format (5.467E+02 5.467D+02 or) or 1 to indicate F-format (546.7).

Different restrictions apply to the input parameters depending on whether the user wants E (D)-formatted output or F-formatted output. These restrictions are discussed next.

33.5.3 E (D)-format Restrictions

To obtain output in E (or D)-format, ts must be 0. The maximum allowed value for the field width fw is 32. If fw is zero, str will be just the length needed, without blanks. If fw is nonzero, fw

must be at least seven and the difference between fw and nd must not be less than seven. (This is because three places are required for the sign, leading digit, and decimal point, and four more for the exponent.) Otherwise, Basis places asterisks (*) in the string. Note that since four characters are always allotted for the exponent, in the case of doubles with a three digit exponent, the D is not printed. Such numbers can not be read correctly by the unformatted string input operator.

Below are some examples and results of calls to format on a workstation when D-format is the desired result.

CALL TO format FOR D-format	**RESULTANT STRING**
str=format(-450.67,14,4,0)	`' -4.5067D+02'`
str=format(-450.67,0,4,0)	`'-4.5067D+02'`
str=format(5.674,8,1,0)	`' 5.7D+00'`
str=format(1.23D123	`' 1.230+123' #No D`
str=format(6,15,1,0)	FORMAT:conversion of integer to string requires exactly two arguments
str=format(4.5,15,1,3)	FORMAT:type specification must be 0 , 1, or 2
str=format(4.5,3,1,0)	`'***'` #field width too small
str=format(4.5,8,3,0)	`'********'` # fw - nd $<$ 7

(Note that on work stations, real literals default to double.)

33.5.4 F-format Restrictions

To obtain output in F-format, ts must be 1. The maximum number of digits which Basis returns is 32. If the field width fw is zero, str is just the length needed, without blanks. If fw is nonzero, fw must be at least 3 and the difference between fw and nd must not be less than 3. Otherwise, Basis places asterisks (*) in the string. If the value being converted is too large to fit in the specified field width, an "r" is placed in the rightmost position of the string. Below are some examples and results of calls to format when F-format is the desired result.

CALL TO format FOR F-format	**RESULTANT STRING**
str=format(-72.4,7,1,1)	`' -72.4'`
str=format(-72.4,0,1,1)	`'-72.4'`
str=format(7654.32145,14,3,1)	`' 7654.321'`
str=format(4.5,2,1,1)	`'**'` # field width too small
str=format(-4.654,5,3,1)	`'*****'` # fw-nd $<$ 3

33.6 Closing File

If a user wishes to close a file, s/he may call the subroutine `basclose`. The form of this call is:

```
call basclose(unit)
```

where `unit` is the unit specifier of the file being closed. It is only necessary for the user to explicitly close a file using `basclose` if the file is currently open as an input or output file, and the user wishes to read that file starting from the beginning. If the user does not want to read an input file more than once, and does not wish to read an output file that has just been created using Basis output commands, then no calls to `basclose` are required.

CHAPTER

THIRTYFOUR

The Macro Facility

Basis has two types of macro definitions. The DEFINE statement is a small abbreviation facility whereas the MDEF-MEND statement is a full fledge macro facility. For either type of macro, a name can be defined as some body of text. Later, when that name is encountered as a token in the input, the body of text is substituted for it and rescanned for tokens. This means that substitution will NOT take place if the word name is inside a quoted string, occurs as part of another name, etc. Another way to keep a macro name from being expanded is to enclose the name within protection brackets { }'s.

34.1 Protection Brackets

The curly brackets { }'s will protect any macro name from being expanded. These brackets can also be used to protect the delimiters in macro calls and COMMAND statements. Protected delimiters will be treated as text and not as delimiters. However, any delimiters not in macro calls or COMMAND statments (such as the commas in a function call) can not be protected.

EXAMPLE:

```
integer x=5
DEFINE x 3
DEFINE title abc
MDEF name = fgh MEND
x
{x}                          ## protect a macro; don't expand it
g command_s {title name}     ## protect macros title and name
g command_s title{ }name     ## protect delimiting blanks in
                             ##   COMMAND statement
MDEF macargs() = some body MEND
macarg({a,b}, c)             ## protect delimiting comma in
                             ##         macro call
```

The results of the above examples will be to

1. print 3 # macro x is expanded to 3
2. print 5 # {x} references the integer x, not the macro
3. call function g with argument “title name”
4. call function g with argument “abc fgh”
5. call macro macarg with two arguments: a,b and c

34.2 DEFINE Statement

```
DEFINE name text
```

defines name to be an abbreviation for the text following up to the end of the line. It should be noted that a semicolon does NOT terminate the DEFINE definition. Rather it is included as part of the definition. This allows you to enter a semicolon delimited statement list in one DEFINE statement.

EXAMPLE

```
DEFINE X y;z
X
```

The above macro will cause both y and z to be printed.

Quotation marks around the definition of the macro are not required, but are allowed. If you wish the definition to be an actual string containing quotation marks, then you would need to double up the quotation marks. Thus the following two lines are equivalent.

```
DEFINE mymacro PLOT  y,x
DEFINE mymacro PLOT "y,x"
```

The following example could help you balance your checkbook:

```
DEFINE check "$-"
DEFINE deposit "$+"
355.66 #opening balance
deposit 433.44
check 55.22
check 12.98
```

is equivalent to the statements

```
355.66
$+433.44
$-55.22
$-12.98
```

which prints out the successive balances desired.

34.3 MDEF - MEND Statement

```
MDEF name = definition MEND
MDEF name () = definition MEND
```

By using the MDEF-MEND statement, you can define macros which allow arguments (up to nine arguments) and can have multiple line definitions. The first form of the macro (the one without the parentheses) is for macro which will never have a parenthesized argument list. The second form must be used if you ever wish to give the macro any arguments. Note: in the MDEF definition, the ()'s must not contain any argument names.

The words MDEF, the macro name, the ()'s if present, and the equals sign (=) must all appear on the same line. The rest of the definition can be spread over as many lines as you like. For example:

```
mdef mymacro =
   y
   plot y,x
mend
```

To reference a macro argument, use the notation $n where n, a digit from 1 to 9, is the number of the argument you wish to reference. If an argument is not present then the value of the argument is a 0 length string.

Besides the $n notation, there are two other macro argument notations: `$*` and `$-`. The notation `$*` refers to the entire argument list—separated by commas, but without parentheses—that the macro was called with. The notation `$-` refers to the entire argument list minus the first argument.

EXAMPLES:

```
mdef addargs() = integer $1 = $2+$3 mend
mdef allargs() = g($*) mend
mdef lessargs() = g($-) mend
mdef someargs() = $1;$2 mend
mdef noargs    = plot y,x mend
                         ## note macro declared without ()'s
addargs(x,5,7)
allargs(arg1, arg2, arg3)
```

```
lessargs(arg1, arg2, arg3)
someargs(arg1, arg2)
someargs(arg1)
someargs
noargs(arg1,arg2)
noargs
```

The above examples will expand to

```
integer x = 5+7                 ## addargs(x,5,7)
g(arg1,arg2,arg3)               ## allargs(arg1, arg2, arg3)
g(arg2,arg3)                    ## lessargs(arg1, arg2, arg3)
arg1;arg2                       ## someargs(arg1, arg2)
arg1;                           ## someargs(arg1)
;                               ## someargs
plot y,x(arg1,arg2)             ## noargs(arg1,arg2)
plot y,x                        ## noargs
```

The expansions of the macros addargs, allargs, and lessargs are straightforward given the definitions of $n, $*, and $-. The expansions of someargs and noargs are a little more complicated.

The expansions of macro someargs shows you what happens when not all the referenced arguments are present. The first expansion of someargs is straightforward. The notation "$1" is replaced by the text "arg1" and "$2" is replaced by "arg2". In the next expansion no second argument is present. Thus "$2" is replaced by a null string, and the resulting body is "arg1;". In the next example no arguments are present. Thus both "$1" and "$2" are replaced by null strings, resulting in a body of ";".

The expansions of macro noargs show you what happens when a macro is declared without arguments. Remember that this macro was declared without parentheses ()'s. Thus a parenthesized list following the macro name is NOT ever considered part of the macro call. Therefore the word "noargs" is expanded to "plot y,x" and the words "noargs(xarg1,arg2)" is expanded to "plot y,x(arg1, arg2)", i.e. the word "noargs" is expanded and the words "(arg1, arg2)" are left as they were found.

34.4 IFELSE Statement

```
IFELSE (arg1, arg2) (arg3, arg4)
IFELSE (arg1, arg2) (arg3)
```

The IFELSE statement takes two argument lists. The first list contains the arguments of the if test. The second list contains the arguments of the if selection. The IFELSE macro is replaced by the text of one of the arguments in the if selection, depending upon the result of the if test.

The two arguments in the if test are first expanded of all macros and then the resulting text of each argument is compared against each other. If `arg1` is identical to `arg2`, then the IFELSE statement is replaced by the text of `arg3`. Otherwise the IFELSE statement is replaced by `arg4` if it is present. If `arg4` isn't present (and `arg1` and `arg2` aren't identical), then the IFELSE statement is replaced by a zero length string, i.e. it expands to nothing.

For example:

```
mdef Dim() = real $1 ifelse ($2, ) ( , ($2))  mend
Dim(x)
Dim(y,100)
```

The above example expands to

```
real x
real y (100)
```

Remember that if an argument is not present, the `$n` notation for that argument expands to nothing. Thus the if test `($2, )` of the above IFELSE statement will determine if the Dim macro was called with a second argument. If a second argument is present then the if test `($2, )` will be false, causing the IFELSE statement to expand to `($2)`. Otherwise the IFELSE statement will expand to nothing.

34.5 UNDEFINE Statement

```
UNDEFINE namelist
```

namelist is a blank and/or comma delimited list of names to be removed from the table of macro definitions. For each name in namelist, the UNDEFINE statement will remove the definition, regardless of whether the macro was originally defined with the DEFINE or MDEF-MEND statements. For worriers: when the words DEFINE, MDEF, or UNDEFINE are seen, macro expansion is turned off so that name is not expanded, thus giving us the chance to see the name so that we can re-DEFINE or UNDEFINE it!).

CHAPTER

THIRTYFIVE

Executing System Commands from the Parser

The user can execute any system or shell command easily from the parser. To do this simply enter as in unix:

! commandline

Example: To give a long list on the Sun of files ending in ’src’:

```
! ls -l *src
```

CHAPTER

THIRTYSIX

Timing

```
TIMER ON | OFF
```

TIMER ON starts a clock running. TIMER OFF prints out the timing statistics since the last TIMER ON. For something fancier, see “TIM: Interrupt Timing” on page 491 of *The Basis Package Library* document. Here is the *OFFICIAL* interface to the system timing routines:

The Fortlib routine timeused has a different number of arguments depending on your system: NLTSS, Sun, UNICOS, ... We hereby publish an *official* interface to the system timing routines:

subroutine ostime(cpu,io,sys,mem) real cpu, io, sys, mem

subroutine glbwrtim(iunit,cpu,io,sys,mem) integer iunit real cpu, io, sys, mem

The glbwrtim routine will print out and label correctly the quantities obtained by ostime. cpu and sys are guaranteed to be in seconds, and represent cpu and system time, respectively. On any system, io is some measure of the io effort, and mem is some measure of the amount of memory resource consumed. The numbers returned by ostime increase monotonically with time. A bigger number is more. That’s all we officially know.

CHAPTER

THIRTYSEVEN

Ending Basis

```
END
quit
quit(1)
```

END terminates the execution of the program. Currently, END must not occur inside a structured statement. The function quit has the same result but may be called from anywhere. If an argument is given for quit, it is used as the exit status.

CHAPTER

THIRTYEIGHT

Error Recovery

In an interpreted language, it is often possible to recover from errors. When an error occurs Basis does its best to help the user understand the nature of the problem. A user variable `debug` governs the error message that goes to the terminal and logfile. The variable `debug` can be assigned the values `yes` or `no`. The default value is no, and error messages will be brief. If `debug` = `yes`, a much more thorough error message goes to the terminal and logfile. If Basis was executing, information is given about the location where the error occurred; a complete trace of all the local variables and arguments to any functions is given, and some symbolic information about the error is given. Whether `debug` = `yes` or `no`, a file is produced that contains the debug information. The filename is first the contents of the variable `probname`, then a numerical extenstion, then the file type appended.

Here is an example. The call to function `boom` fails when it attempts to add two arrays that are not the same length. The error message, "Operands not compatible in size for +" is followed by a more detailed error analysis because `debug = yes`.

```
Basis> function boom(a,b)
Basis> chameleon temp
Basis> temp= (a+b)/2
Basis> return [temp,temp**2]
Basis> endf
Basis> debug=yes;boom([2,2],[2,3,4])
parcnfm: Operands not compatible in size for +
Writing traceback info to file problem.001
Returned to user input level.
```

The relevant contents of the file problem.001 are:

```
Here is the information I have on where you were:
   A call to boom containing
   the problem.
The error occurred in the assignment or append statement:
               temp = expression
The following lines contain clues(not facts) about the r. h. s.
```

```
b
+
a+b
temp
Parser’s action number = 115(ADD), program counter = 45.
Group: Locals_boom  Num Vars: 3
a(2)
 1:     -  2 2
b(3)
 1:     -  2 3 4
temp     =  0
```

If the parser function `errortrp("off")` has been called, no error recovery is attempted.

A compiled routine `kaboom(iflag)` can be called from the parser to force a return to the parser. If iflag is greater or equal to 0 Basis acts as if an error has taken place, and produces a trace file. If iflag is set negative, Basis returns to the parser without any error messages.

```
function subt(a,b)
if( a < b) then
     remark "Error: subt called with a < b"
     call kaboom(-1)
endif
return a-b
endf
```

In interpreting the information printed out when `debug` = `yes`, you should begin with the error message itself, examine the description of the nesting levels to find out where you were generally, and examine the symbols listed for some hints about the parts of the expressions involved in the error. As a last resort, the pc counter can be used in conjunction with the list command. The value of the pc counter is given in the `trace` file. Do a list on the function in which the error occurred, and when asked answer ‘y’ to the question about viewing the intermediate code. The listing which results shows the operations being performed and using the pc you should be able to pinpoint which instructions caused the error. For example, in the example above, boom reported that the error occurred at pc = 45.

```
Basis> list boom
boom(a,b)
    user-defined function
Minimum number of arguments:     2
Maximum number of arguments:     2
User-defined function, begins at absolute address      3967064
Function consists of    104 words of intermediate code.
 NAME                    TYPE
 boom                      varies
```

```
 a                          varies
 b                          varies
Dump intermediate code? (y|n)
y

   pc   opcode  stack     operation
    1:     16          Enter function, set up actual parameters.
    3:    421      7   REGULAR_SCOPE
    5:    404      9   CHAMELEON
    7:   -100     10   TOKEN = 'temp' (name).
   12:    412     10   SCALAR DECLARE
   14:    415     10   NO INITIAL VALUE
   16:    410     11   CREATE VARIABLE
   18:   -100      8   TOKEN = 'temp' (name).
   23:      1      8   ID->LHS
   25:     10      9   BEGIN RHS
   27:   -100     11   TOKEN = 'a' (name).
   32:      1     11   ID->LHS
   34:    136     11   <LHS>-><FACTOR>
   36:   -100     13   TOKEN = 'b' (name).
   41:      1     13   ID->LHS
   43:    136     13   <LHS>-><FACTOR>
   45:    115     13   ADD
   47:    130     12   MOVE-1
   49:    136     10   <LHS>-><FACTOR>
   51:   -101     12   TOKEN = '2' (integer constant). value = 2.
   56:     11     12   PUSH VALUE

   58:    121     12   DIVIDE
   60:      3     10   ASSIGN
   62:   -100     10   TOKEN = 'temp' (name).
   67:      1     10   ID->LHS
   69:    136     10   <LHS>-><FACTOR>
   71:      8     10   FETCH VARIABLE
   73:   -100     12   TOKEN = 'temp' (name).
   78:      1     12   ID->LHS
   80:    136     12   <LHS>-><FACTOR>
   82:   -101     14   TOKEN = '2' (integer constant). value = 2.
   87:     11     14   PUSH VALUE
   89:    126     14   POWER
   91:    101     12   EXPLIST
   93:    129     11   [EXPLIST]
   95:    136      9   <LHS>-><FACTOR>
   97:     19      9   FETCH COPY
   99:     17      9   RET
```

```
  101:     18      8   NULLRET
  103:     17      9   RET
**************************
```

CHAPTER

THIRTYNINE

Interrupting Basis

Basis can be usually interrupted by typing control-C. The terminal interactions of the operating system sometimes make it hard to do this if a lot of output is being displayed and several tries may be required. Basis checks for this message before each step of intermediate code and before most lines of output. The routine `ruthere` looks for the control-C message. If you are executing in a compiled routine the interrupt may not work unless the author has inserted "`call ruthere`" in the program at strategic points. Annoy your author until he or she does so. However, if the `ctl` package `run` command or its relatives are controlling the operation of a physics package, a check is made after each step and authors need not include `ruthere` calls in that case.

CHAPTER

FORTY

List of Reserved Words

Basis reserved words are written in upper case throughout this manual for purposes of emphasis, but are recognized by Basis if they are entered entirely in lower case. These words cannot be used as identifiers:

```
BREAK, CALL, CHARACTER, COMMAND, COMPLEX, DEFINE, DO,
DOUBLE, ELSE, ELSEIF, END, ENDDO, ENDF, ENDFOR,
ENDIF, ENDWHILE, FOR, FORGET, FUNCTION, IF, IFELSE,
INDIRECT, INTEGER, LINLIN, LINLOG, LIST, LOGICAL,
LOGLIN, LOGLOG, MDEF, MEND, NEXT, OUTPUT, PLOT,
PLOTM, RANGE, READ, REAL, RETURN, THEN,
UNDEFINE, UNTIL, WHILE.
```

CHAPTER

FORTYONE

List of Non-Alphanumeric Tokens

!	+	<	[
%	,	<=	]
&	-	<>	—
’	.	=	~
(	/	==	~=
)	/!	>	(space)
*	//	>=	(return)
*!	{	?	‘
**	}	@	:
;	^	—=	&=
+=	-=	*=	/=
**=			

Here is a list of the non-alphanumeric tokens used in basis. Any input character other than one of these, which is not alphanumeric, will cause an “illegal character in input” error (unless, of course, it is part of a comment).

CHAPTER

FORTYTWO

List of Parser Variables

42.1 Variables

asgnchek controls whether or not the limiting string of a variable is used in determining the bounds of an array which is the target of an assignment statement. The default value is `yes`. Use `asgnchek=no` to allow storage to array elements outside the current `setlimit`'ed values.

autocr If = `yes`, then each output command, `<<`, will automatically supply a carriage return. If `autocr = no`, then no carriage return is automatically supplied by Basis. Output is buffered until either a RETURN is included in an output command or the buffer is exceeded. Default = `yes`.

autodyn when `autodyn = yes` any attempt to access a dynamic array will cause storage to be allocated for it if it does not already have it. There are two auxiliary variables which are used when this occurs: `autodynp` is an integer containing an amount of padding to be added to such arrays, and `autodyna`, if set to a non-blank string, will give that attribute to any array allocated in this way. These two variables default to 0 and blank, respectively. The default value of `autodyn` is `no`.

autohist Controls the handling of display statements. If it is set to 0, each displayed quantity is assigned to the variable `$` and then displayed. This is the default.If `autohist <> 0`, the displayed quantity will be assigned to one of the 26 variables `$a`, `$b`, ...,`$z` depending on `mod(autohist,26)`, with `$a` corresponding to `autohist = 1`, `$b` to `autohist = 2`, etc. Then `autohist` will be incremented. So, if you set `autohist = 1` to begin with, the results printed will be saved in `$a`, `$b`, etc., and after `$z` back to `$a`. Thus a "history buffer" of 26 previous results will be maintained.

autovar when `autovar = yes` undeclared and unsubscripted variables are automatically declared. Default is `no`.

compress if `yes`, compress on output repeated array elements; if `no`, list each element of an array separately. For more extensive control see "The Stream I/O Facility" on page 143.

cprompt can be set to any character string up to length 16 to change the basic prompt.

debug is set to `yes` or `no` to control the amount of detail in the error printout. `debug` = `yes` causes extensive printouts; `debug` = `no` does not, but the program executes much faster. The default value is `no`.

dec toggle the output to occur in decimal form. The default.

debuga used to control detailed debugging printouts

debugc If `yes`, print stack dump before and after each action.

echo is set to `yes` (1), `no` (0), or `logonly` (2) to control echoing of lines from input files. Default is `yes`. If `echo` is `logonly` then lines are echoed into the log but not to the terminal.

eof an integer which contains the value `no` if the last read attempt was successful, and `yes` if the last read attempt was unsuccessful.

fuzz number of digits after decimal point in prints. Default = 5. For more extensive control see "The Stream I/O Facility" on page 143.

hex toggle the output to occur in hexadecimal form. Decimal (`dec`) is the default.

coredump if `yes`, dump core when exiting if the exit status is not 0. Default value is `no`. An obsolete but still working alias for `coredump` is `keepdrop`.To disable the system's error recovery, type: call `errortrp`("off"). To restore error recovery type: call `errortrp`("on").Exit status is set non-zero when exiting because error recovery is off or if routine `quit` is called with a non-zero argument.

lcprompt should be set to the number of characters in `cprompt` that are to be used.

lsprompt should be set to the number of characters in `sprompt` that are to be used.

noisy If set to `no`, ignore all non-numeric tokens ("noise") in stream input. If set to `yes`, all tokens are significant and are to be assigned to a corresponding input stream variable. Default is `no`.

notty If `yes`, terminate the run after processing the macfiles. Otherwise go on to the Basis prompt. Default is `no`.

nskipr n If n $>$ 0 skip the first n records on the next file read. Basis automatically resets `nskipr` to 0.

oct toggle the output to occur in octal form. Decimal (`dec`) is the default.

padding Each call to `allot` or `change` allocates a certain number of elements. To this amount, `padding` extra elements will be added. The extra space is not used by Basis in any way.

sprompt can be set to any character string up to length 16 to change the secondary prompt.

switches An array of 100 real switches. Defaults to 0.

verbose If set to `yes`, print out all the system messages to the TTY and the log file. Default is `yes`.

42.2 Constants

blank is a 80-character variable full of blanks

false contains the logical constant .false.

off contains “off”; many packages expect “on” or “off” as the setting for devices or plot options.

on contains “on”; many packages expect “on” or “off” as the setting for devices or plot options.

no contains an integer 0; many packages expect `yes` or `no` as values for switches, such as `echo` above.

pi contains the real value of pi (3.14159...).

stdin contains the unit number for reading from the terminal.

stdout contains the unit number for writing to the terminal.

stdplot contains the unit number for writing to the plot file.

true contains the logical constant .true.

yes contains an integer 1; many packages expect `yes` or `no` as values for switches, such as `echo` above.

CHAPTER

FORTYTHREE

List of Compiled Functions

This section describes functions that are callable from the Basis Language. These routines are all ordinary compiled Fortran and you can pass them arguments by value (the default) or by address (using the form `&x`). (see “Built-in Functions” on page 101.)

43.1 Working With Attributes

Each named entity (variables, functions, and macros) can have zero or more “attribute” words associated with it. These words are then available as keys to select names on which special functions called “attribute servers” can operate. Normally attributes are given to variables by program authors. Users may give or remove attributes using the function rtcattr:

```
call rtcattr("name","attributes")
```

Here attributes is a space delimited list of attribute words (up to 24 characters). A word can be prefixed with a minus sign to remove an attribute from a word.

The existence of an attribute can be tested with the function rtattr:

```
iflag = rtattr("name", "attribute")
```

`iflag` will be `TRUE` if the attribute exists and `FALSE` otherwise.

Writing attribute servers is explained in the document Writing Basis Programs: A Manual For Program Authors. See “Writing Attribute Services” on page 425. There are two servers built in to Basis:

```
call attrlist(aexp,iunit)
call attredit(aexp,iunit)
```

Here `iunit` is a unit number, and `aexp` is a quoted string containing a logical attribute expression. A logical attribute expression is built up from attribute names, parentheses, and the operators

& (and), | (or), and ~ (not). Plus and minus can be used as synonyms for | and -. The routines respectively list, or edit the variables whose attributes satisfy `aexp`, to the unit iunit.

If a user wants to print the values of certain variables, for example, she might call `rtcattr("name","mylist")` for each variable name desired. Then `attredit("mylist",stdout)` will print all such variables to the terminal.

43.2 Help and News

Subroutines `help` and `news` supply information about the help package and the most recent changes, respectively.

43.3 Memory Management of Dynamic Arrays

The following routines are more thoroughly documented in Writing Basis Programs: A Manual For Program Authors. See “Writing Basis Packages” on page **??**.

`call allot("array",length)` allocates an array of length elements. The quotes around the name are required. If array is a multidimensional array, length is the length of the desired last dimension of array. The database manager knows the type and other dimensions of array. The package to which array belongs is determined by the current context. Each element would contain 2 words if array is complex.It is not an error if array has already been allocated space that has not been released by a call to basfree.

`call basfree("array")` Releases space for array previously obtained by a call to `allot`. The quotes around the name are required.

`call change("array",newlength)` changes the length of array to `newlength`. The quotes around the name are required. The comments above about multidimensional arrays apply here as well.

`call gallot("name",n)` allots all the dynamic arrays in group, name.

`call gchange("name",n)` changes the allocation of all the dynamic arrays in group, name.

`call gfree("name")` frees all the dynamic arrays in group, name.

43.4 Opening and Closing Files

`integer basopen`

`iunit = basopen(name, access)` This routine is used for opening input files and for creating output files.If access is “r”, opens file `name`, returning the unit number to use in subsequent operations. If the file is not present, it is searched for (using the list in variable `path`, which can be added to with the variable codefile in config, or by the routine pathadd). Error recovery is invoked if the file cannot be found at all.If access is “i”, `basopen` returns OK or ERR (0 or -1) to indicate whether or not the file can be opened in “r” mode.If access is “w”, the file is created, returning the unit number to use in subsequent operations. Error recover is invoked if the file cannot be created.Any file opened with `basopen` will be CLOSED whenever error recover takes place.

`call outfile(&j,comment)` opens a text output file and places the value of the unit specifier into the integer variable `j`. Note that since `j` is an output quantity it must be passed by address. `comment` is a character string that is used to comment the file. When Basis terminates, if `verbose` is yes, the files created by `outfile` will be listed along with the comments supplied. Basis generates the name of the file from a combination of the value of `probname`, and a counter. Files created with outfile are NOT closed when an error occurs. If comment = “*temporary*”, the file is deleted when the program terminates. `basclose (unit)` should be used to close files opened with outfile.

`call basnxtsq(f,g) (Fortran)`

`g = basnxtsq(f) (Basis)` given a filename f sets g to the next name in the sequence when called from Fortran, or returns the next name in the sequence when called from Basis. In Fortran, f and g may be the same variable. Some of the sequence types handled by basnxtsq, using an algorithm of Dave Munro’s, are: prob01fa prob01fb prob01fc prob01fd ... prob02q00 prob02q01 prob02q02 prob02q03 ... prob02q43 ... prob03q00.pdb prob03q01.pdb prob03q02.pdb prob03q03.pdb ... prob03q00.cdf prob03q01.cdf prob03q02.cdf prob03q03.cdf ...

43.5 Executing User Functions

You can execute a user function `f` by

`call execuser("f")` The function `f` must have no arguments and cannot return a value. This function is usually called from compiled code.

43.6 Adding Comments to Variables and Functions

`call comment("name","comm")` adds a comment to any variable or function in the database including those defined by the user.

43.7 Checking for the Existence of Variables and Functions

You can check whether a variable or function has been defined by typing:

```
if(exists("name"))
```

43.8 Flushing the LogFile

`call flushlog` flushes the log file. This can be useful after an error recovery if some vital information has scrolled off your screen.

43.9 Using the Switches Array

You can set switches(i) = x if you type:

```
call swset(i,x)
```

You can get the value of switches(i) by invoking the function switch:

```
switch(i)
```

43.10 Protecting User-Defined Variables and Functions

You can protect user-defined functions and variables made up to this point from future FORGETs by typing:

```
call protect
```

43.11 Setting Variable Dimension Limits

`setlimit("name", "( dimension )" )` allows you to restrict the portion of an array that will be used. The function setlimit can be called from user or compiled code. Setlimit uses the usual search to determine the meaning of name. The parentheses in the second argument are required. The restrictions on dimension are the same as for regular dimensioning strings: the contents of the string must consist of constants, operators, and names which can be evaluated. The allowed operators are `+, -, *, /`. In evaluating names in this string, the database for name is searched first; only if this fails is the usual search made.Subsequent accesses to this array cause a reevaluation of the limiting string so that changing variables

which appear in the limiting string will change the amount of the variable used.The limiting string must define the correct number of dimensions and the values for the limits must be within the storage dimension values. Exception: the upper limit for any dimension may be set to one less than the lower index of the array, thus declaring that no part of the array is currently in use.

`setlast("name", n)`

`rtadddim("name")` The routine `setlast("name", n)` limits the LAST dimension (only) of the variable name to a high subscript of n. If n is greater than the current highest (unlimited) last subscript of name, then an attempt is made to expand storage so that n will be a legal subscript. The size of the last subscript is increased to the maximum of n and its current value times 1.5 with a minimum of 16. `setlast` can be used on static arrays as long as no attempt is made to exceed the actual storage available.The routine `rtadddim("name")` adds a dimension to the variable name, which is sometimes useful in conjunction with setlast and the append statement.

43.12 Specifying Assignment Actions

For each variable, the user may specify a string containing Basis language statements called its assignment-action string. This string will be parsed and executed after each assignment statement in which the corresponding variable name appears on the left-hand side of the assignment statement. To set the string do:

call `setact("name", "action" )` Any subsequent assignment statement which changes the variable `name` will cause the action string to be parsed and executed. Restrictions are: the action string must consist of complete statements (e.g., compound statements like do loops are fine but must be complete); the action string must be 72 characters or less; the action should not induce an infinite recursion (such as `real w;setaction("w","if(w<0) w=-1.")`). Examples: `real x; call setact("x","x")` will cause the value of x to be printed whenever it is changed with an assignment statement. (Useful for debugging!). If y is a parameter which is supposed to contain a value between 0. and 1., an author might do something like: `call setact("y", "if(~ (y ? [0.,1.])) then; remark ""Bad y"";kaboom(0);endif")` to prevent the user from assigning a bad value.Since the action may include a call to any function, the restriction on the length of the string can be easily finessed.

43.13 Redefining Array Shapes

`setshape("name", "( dimension )" )` allows you to reset the dimension statement of a variable. It does not change the storage allocated, merely the perceived shape of the array. If the number of items in the new dimension does not equal the current number of

items an error is issued. See `setlimit` above. The function `setshape` can be called from user or compiled code. `setshape` uses the usual search to determine the meaning of name. The parentheses in the second argument are required. The restrictions on dimension are the same as for regular dimensioning strings: the contents of the string must consist of constants, operators, and names which can be evaluated. The allowed operators are `+`, `-`, `*`, `/`. In evaluating names in this string, the database for name is searched first; only if this fails is the usual search made.

`useshape("name")` evaluates the existing symbolic dimensions and assigns the shape to the variable. This is useful when a variable already has a symbolic shape that is assigned by `rtvare` but the memory is allocated by the client code. This call has the effect of causing Basis to use the current shape of the memory. The function `useshape` can be called from user or compiled code. `useshape` uses the usual search to determine the meaning of name.

43.14 Functions With Variable Numbers of Arguments

If the parameter list of a user-defined or compiled function contains a semicolon at the beginning or in place of one of the normal commas, the arguments which follow the semicolon are optional. The function can be called with none, some, or all of its optional arguments.

Additionally, you can use the function `setmnarg` to declare optional arguments for both user and compiled functions.

`setmnarg("name",n)` sets the minimum number of arguments to the function `name` to be `n`. The function must be a user or compiled function that has at least `n` arguments.

When a user calls the function without all of its arguments, what happens depends on whether the function is user-defined or compiled.

For user-defined functions, the local variables corresponding to the arguments which were not supplied are simply not created, which will cause an error if an attempt is made to access that name *unless* a variable with the same name as the formal parameter exists in the search path. This can be used to set default arguments. For example, if I have a physics variable named gamma which is the usual third argument to a function f, then I can write f as follows:

```
function f(a,b;gamma)
.....
endf
```

after which f(1.,3.) results in the physics variable gamma being used as the third argument, while f(1.,3.,5.) uses 5. as the third argument. This works because when the call f(1.,3.) occurs, no variable named gamma gets created in the local variables for f, and so references to gamma in f become references to the gamma in the search stack.

A macro `default` is built in to Basis which makes it easy to supply default values locally. The usage is:

```
default(name) = value
```

If the `= value` portion is omitted, name will be created as an integer scalar with value 0 if the argument name is not supplied. If the `= value` portion is given, and the argument name is not supplied, name is created as a chameleon variable and value is assigned to it.

For compiled functions, Basis will fill in the missing arguments by passing one of the following values, depending on the type of the argument:

type	**default value**
integer	DEFAULT
real	float(DEFAULT)
complex	(float(DEFAULT), float(DEFAULT))
logical	FALSE
character	rone blank

The constant DEFAULT is supplied by MPPL for authors to use to decide if an optional argument was omitted or not.

43.15 Creating Pauses

`call paws` which causes Basis to pause and request a carriage return to continue. If the user sends any other message an error exit is taken through `kaboom`.

43.16 Returning to the Parser

You can force a return to the prompt with the statement:

```
call kaboom(iflag)
```

If `iflag <> 0 & debug=yes`, this can create a long very useful printout.

43.17 Recursive Parsing

You can compile and execute a statement in the Basis Language with the subroutines `parsestr`, `parselng`, and `parse`:

```
character*(n) s   # n a number <=500
character t(m)
character*(n) u
```

```
call parsestr(s)  #or,
call parselng(t,m) #or,
call parse(u,n)
```

Restriction: the string to be parsed cannot contain a READ statement.

These routines can be called from anywhere within the Basis environment: from the interpreter, from a compiled routine (for instance, in a physics package), from a built-in routine, or even through some"hook" in a graphics library. It may be called recursively to any depth; for instance, it may be asked to parse a string which itself contains a parsestr call. Each time it is called, it saves sensitive portions of its environment on a stack, thus making this flexibility possible.

Thus,

```
call parsestr("global real x = 3.")
```

will execute the statement "global real x=3.", creating a real variable x whose value is 3.

Any variables created without the "global" scope are created in the stack frame of the parsestr function itself, and will therefore disappear on return. Any user-defined functions declared will be defined after return. If the string does not represent a series of complete statements those statements not yet completed are discarded without execution. If a syntax error or semantic error occurs, error recovery occurs as usual and one is returned to the bottom level parser.

The maximum length of strings is a system-dependent limit (about 500 on Crays).

The routine `parselng` allows you to exceed this by using a character array. And the routine `parse` allows you to pass an array of strings which you wish treated as a sequence of lines; `parse` will insert semicolons between the array elements and pass the result to `parselng`.

43.18 RANF and Its Supporting Routines

Ranf is a 48 bit multiplicative congruential method RNG which produces 64 bit floating point numbers in the open interval (0,1). More precisely, it produces a sequence of uniform variates U_i based on the following formulas:

$$U_i = \frac{S_i}{2^{48}} \tag{43.1}$$

$$S_{i+1} = aS_i \bmod 2^{48} \tag{43.2}$$

where the (integer) multiplier a = 0x2875a2e7b175 [1] and the (integer) default seed S_0 = 0x948253fc9cd1. Note that the minimum value for U_i is $1/2^{48} \cong 10^{-15}$ and the maximum value is $1 - 1/2^{48}$. [2]

[1] The multiplier's inverse is a' = 0x5ceeb894d6dd, with $aa' \equiv 1 \bmod 2^{48}$.

[2] If stored in an IEEE 754 Standard single precision (32 bit) floating point format, the minimum is distinct from 0; however, the maximum (and many other values near it) are not distinct from 1.

On the Cray, *ranf* is loaded from the *Mathlib* library. The workstation version is based on the *drand48* suite of library functions. Although these two libraries implement the same basic arithmetic, there is a subtle difference in that *drand48* computes the next seed and returns that value divided by 2^{48}, whereas *ranf* saves the old seed, computes the next, and then returns the old seed divided by 2^{48}. The result is that the sequence from *drand48* is "one ahead" of that from *ranf*. This problem may be solved by decrementing or incrementing the seed by one as part of setting or retrieving it, respectively, and this logic is built into the workstation versions of *setranf* and *getranf* routines below. Thus sequences generated by the Basis *ranf* are identical on Cray or workstation, and seed values may be carried between the two architectures without a break in the sequence.

The examples below are written as they would appear in source to be preprocessed by *Mppl*. In Fortran terms, *ranf* will return *double precision* on a 32 bit workstation, and *real* on a 64 bit architecture.

43.18.1 Ranf

```
real(Size8) ranf,x
x = ranf(0)
```

Ranf may also be called from the Basis interpreter as a built-in function.

43.18.2 Getranf

```
integer iseed48(2)
call getranf(iseed48)
```

Getranf reads the current 48 bit seed, placing the lower 32 bits in *iseed48(1)* and the upper 16 bits in *iseed48(2)*. It may be called from the Basis interpreter, but be careful to pass its argument "by reference" in that case: *call getranf(&iseed48)*.

43.18.3 Setranf

```
integer iseed48(2)
call setranf(iseed48)
```

Setranf restores a 48 bit seed (presumably stored in *iseed48* by an earlier call to *getranf*). If both elements of the array are zero, the default seed is reset. *Setranf* may be called from the Basis interpreter as a compiled function.

43.18.4 Seedranf

```
integer iseed
call seedranf(iseed)
```

The semantics of *seedranf* are similar to Cray Mathlib's *ranset*, with some restrictions. If *iseed=0*, the default seed value is restored. Otherwise, the given value (or the next odd integer if *iseed* is even) is set as the current seed. If you're setting an arbitrary seed, be aware that integers on the workstation are usually limited to 32 bits, and that the upper 16 bits of the 48 bit seed are set to zero by this call. Thus, the first value returned by *ranf* will be quite small (

).*Seedranf* is provided primarily as a convenient way to reset the default seed - e.g., "*call seedranf(0)*" This function is also available within the Basis interpreter.

43.18.5 Mixranf

```
integer iseed, iseed48(2)
call mixranf(iseed,iseed48)
```

Mixranf provides functionality similar to Mathlib's *rnfmix*: If *iseed*¡0, the default seed is set. If *iseed*=0, then a "random" seed is created from the system clock plus 10 calls on *ranf*. If *iseed*>0, then the value is set directly as per *seedranf*, with similar wordsize restrictions. *Mixranf* can be called from the Basis interpreter.

43.19 Manipulating the External Environment

These functions do some (simple) things that are otherwise hard to accomplish inside Basis programs. Here's what's currently available:

43.19.1 basisexe()

NAME

`basisexe()` - execute a shell command

SYNOPSIS

```
integer status
status = basisexe("ls ~/wrk")
```

DESCRIPTION

The `basisexe` function may be executed either from FORTRAN code or directly from the command line at runtime (although use of the shell escape '!' requires less typing). The argument should be a quoted string containing a legal shell command (or commands, separated by semicolons). This function returns the status code of the command (normally zero unless there was some error).

43.19.2 cd, chdir()

NAME

`cd, chdir()` - change working directory

SYNOPSIS

```
cd /foo/bar
logical chdir("/foo/bar")
```

DESCRIPTION

The `cd` command works almost exactly like its counterpart in the UNIX shells. It accepts the standard shorthand meaning for the tilde "~" character. `cd` with no arguments changes to your `HOME` directory. For use in scripts, the `chdir()` function is provided. It returns logical true or false depending on whether the command succeeded or failed. Typical use:

```
    if(chdir("/foo/bar")) then
    remark "it worked!"
else
    remark "try something else"
endif
```

43.19.3 setenv, getenv

NAME

`setenv, getenv()` - set or read environment variables

SYNOPSIS

```
setenv foo bar
character *64 homedir = getenv("HOME")
```

DESCRIPTION

`setenv` works just like its counterpart in the C shell, and is occasionally convenient for resetting, say, `NCARG_ROOT` from within a running Basis program. The `getenv()` function (which formally returns "character *(500)") is provided for symmetry, and to make it easier to set a Basis variable to the value of a given environment variable. For example,

```
chdir(getenv("PWD"))
```

will usually set the working directory to the value it had when you started a Basis session.

43.19.4 diskspace

NAME

`diskspace()` - find the remaining free space in your file store

SYNOPSIS

```
real xxx = diskspace("/foo/bar")
```

DESCRIPTION

The `diskspace()` function returns the number of megabytes of space available to you in the filesystem containing its pathname argument. If no argument is given, the current working directory is assumed. Diskspace attempts first to determine if you have a quota assigned on the filesystem in question. If you do, it returns the amount of free space available before you hit your hard limit. If you don't have a quota, it simply returns the available free space on the disk, just like the "df" command. It returns -1 on error. Typical usage:

```
    cd ~/wrk
if(diskspace() < 20.0) then # Quit if less than 20MB free
    fin
else
    remark "Running another cycle"
endif
```

Part III

EZN User Manual: The Basis Graphics Package

CHAPTER

FORTYFOUR

Introduction to EZN

44.1 Essential Setups and Simple Experiments

EZN is a Basis package which supplies a user interface to the National Center of Atmospheric Research (NCAR) Graphics Library (see http://ngwww.ucar.edu). The EZN package is a standard part of the programs `Lasnex` and `Sod`. The EZN package has an additive model, that is, each plot command you issue adds something to the picture until you issue a `nf` ("new frame") command. The package contains curve, marker, contour, and text commands with facilities for titles, frame control, and viewport control. It also contains some commands which use Lasnex-specific data structures, such as mesh and mesh-based contour commands.

EZN works with the Graphics Kernel System (GKS) which comes with NCAR. The current version of the package requires NCAR4.0.1 or later. 45See CHAPTER 3: "Devices" on page 15 for information on the graphics devices supported by EZN.

Before using a program containing EZN, make sure the Basis environment variables are set correctly. Refer to Section 1.1 "Environment Variables" in Chapter 1 for the list of environment variables needed.

The most effective way to learn EZN is by doing some simple experiments. If you are executing one of the programs containing the EZN package (Basis, Sod, etc.), you can enter the following examples at your terminal to see how EZN works. These assume that you are able to display X windows on your terminal. Any text following a "#" is a comment. We set the variable `ezcshow` to `false` so that the pictures will not appear until the `nf` command is given. If you want to see the picture before it is completed, you can issue a `sf` or "show frame" command at any time.

```
ezcshow=false   #Don't show pictures until "nf" command given.
win on  # Open an X window on your workstation.
cgm on  # Open a CGM file to record the pictures.
```

The following statements set up values for variables used in later commands.

```
# Calculate some data to be used in the examples:
real x(20)=iota(20)
real y1=x**2, y2=x**2.1, y3=x**2.2
```

```
# Example 1
titles "Example 1","One Curve","Y1"
plot y1 x
sf  # Show the frame.
```

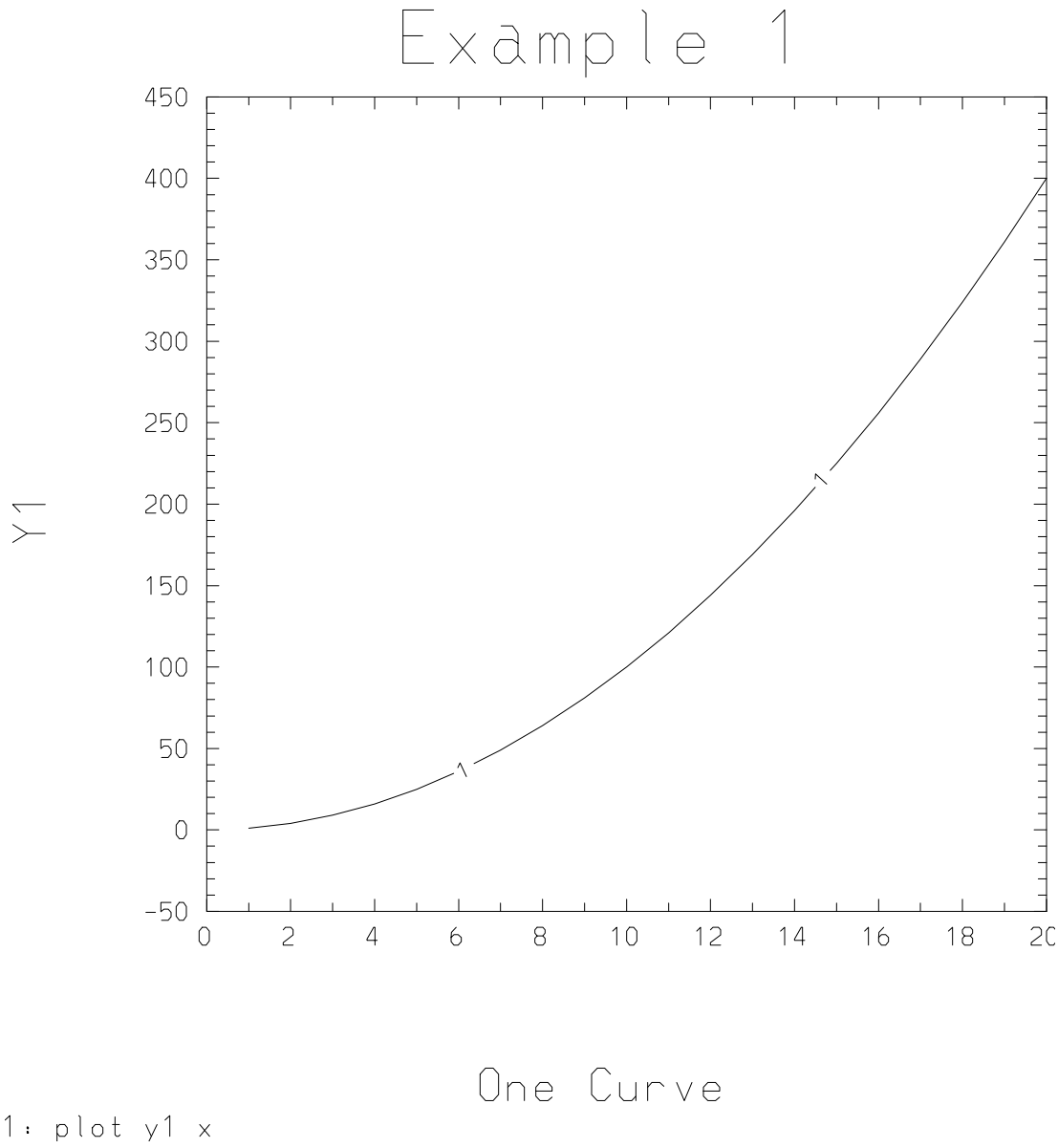


Figure 44.1: Example 1

```
# Example 2
# Change the thickness, color of the plots.
nf  # Clear display list for next example.
titles "Example 2","One Curve, Thick and in Red","Y1"
plot y1 x color=red thick=2.
sf
```

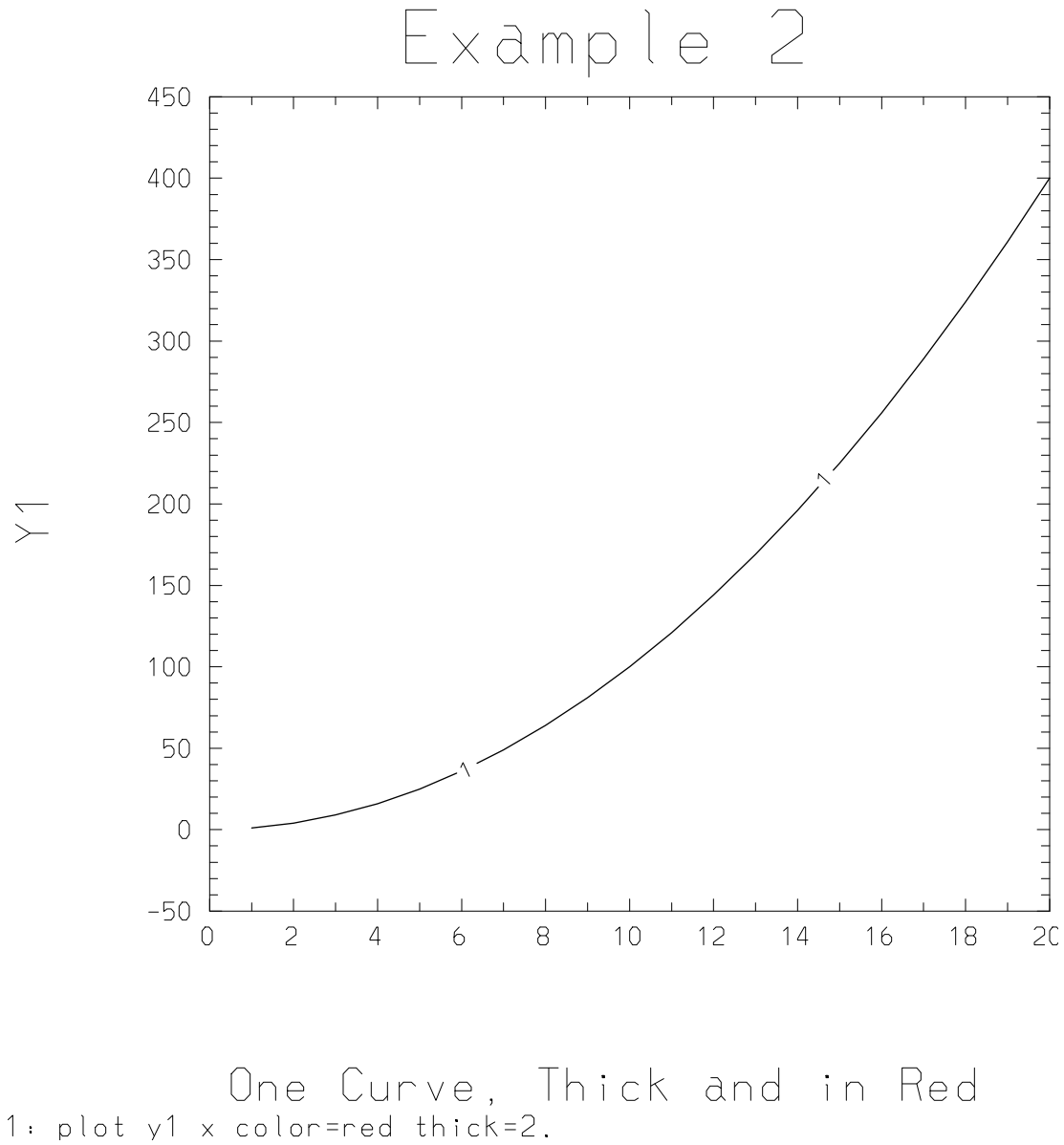


Figure 44.2: Example 2

```
# Example 3
nf
titles "Example 3","Three Curves on One Plot"
plot y1 x color=red
plot y2 x color=blue style=dashed
plot y3 x color=green style=dotted
sf
```

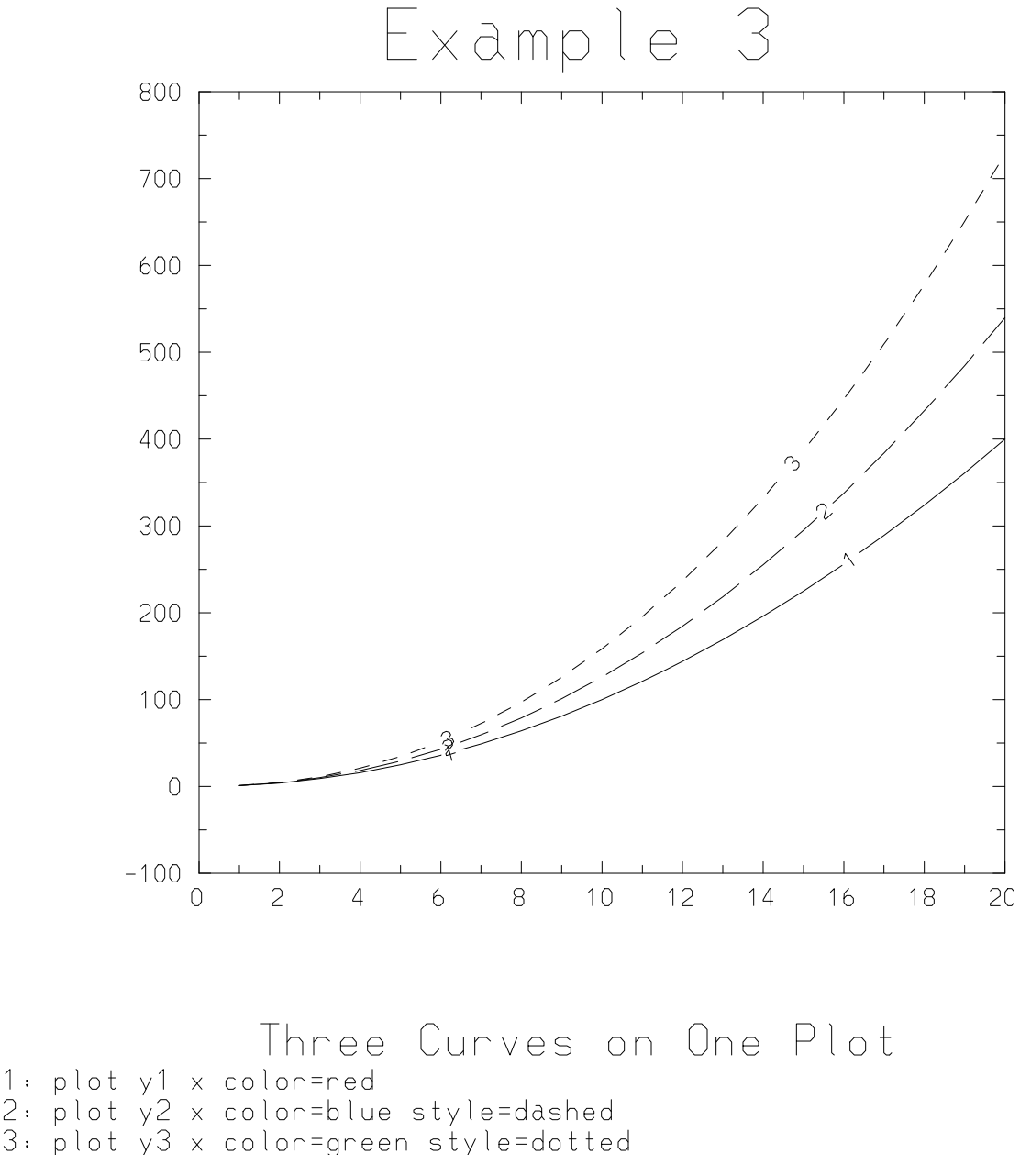
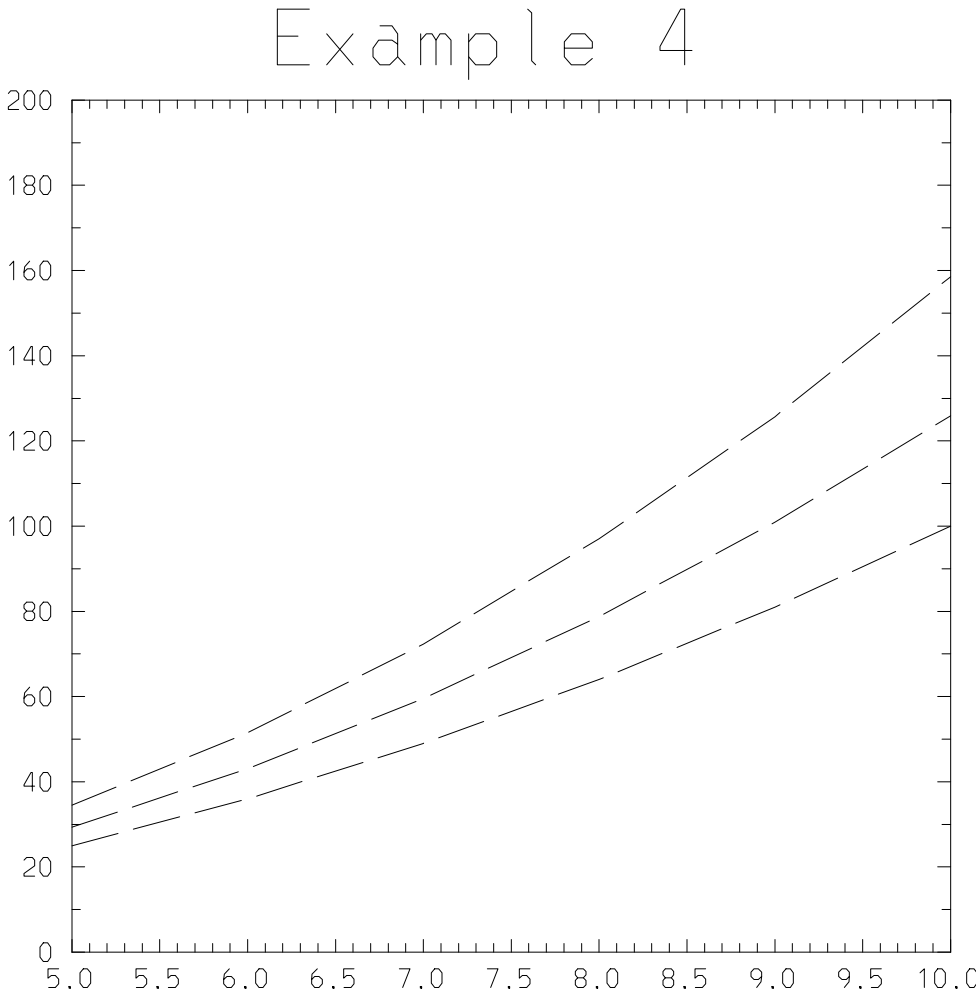


Figure 44.3: Example 3

```
# Example 4
nf; titles "Example 4",
   "Three Dashed Curves on One Plot, Frame Set, No Labels"
frame 5. 10. 10. 200.
attr labels=no style=dashed
plot y1 x color=red
plot y2 x color=blue
plot y3 x color=green
sf
```



Three Dashed Curves on One PLot, Frame Set, No Labels

```
plot y1 x color=red
plot y2 x color=blue
plot y3 x color=green
```

Figure 44.4: Example 4

```
# Example 5
nf
titles "Example 5","Three Curves on One Plot, Labeled"
# Note that we don't have to keep repeating x:
plot y1 x color=red labels="a"
plot y2   color=blue labels="b"
plot y3   color=green labels="c"
sf
```

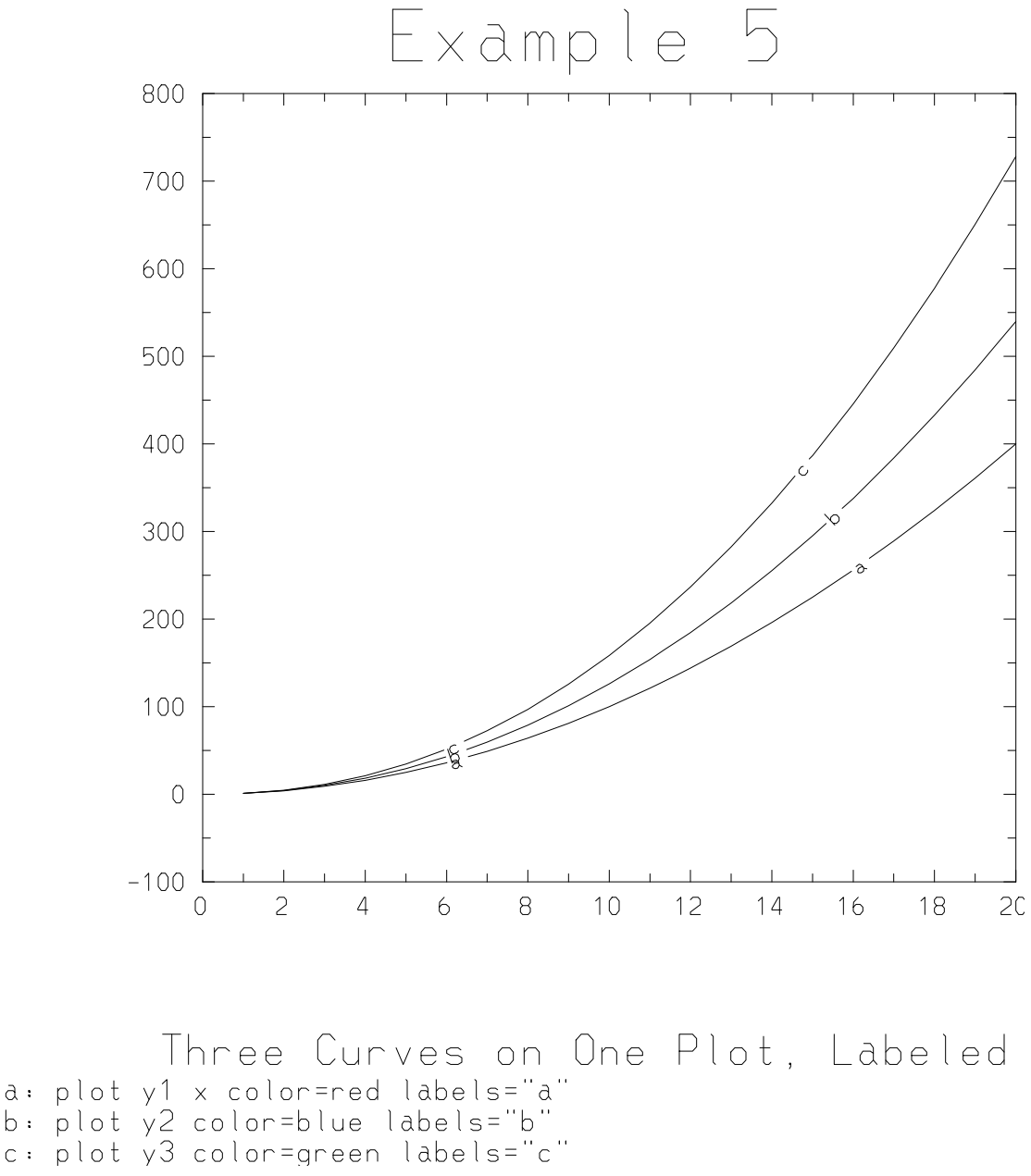
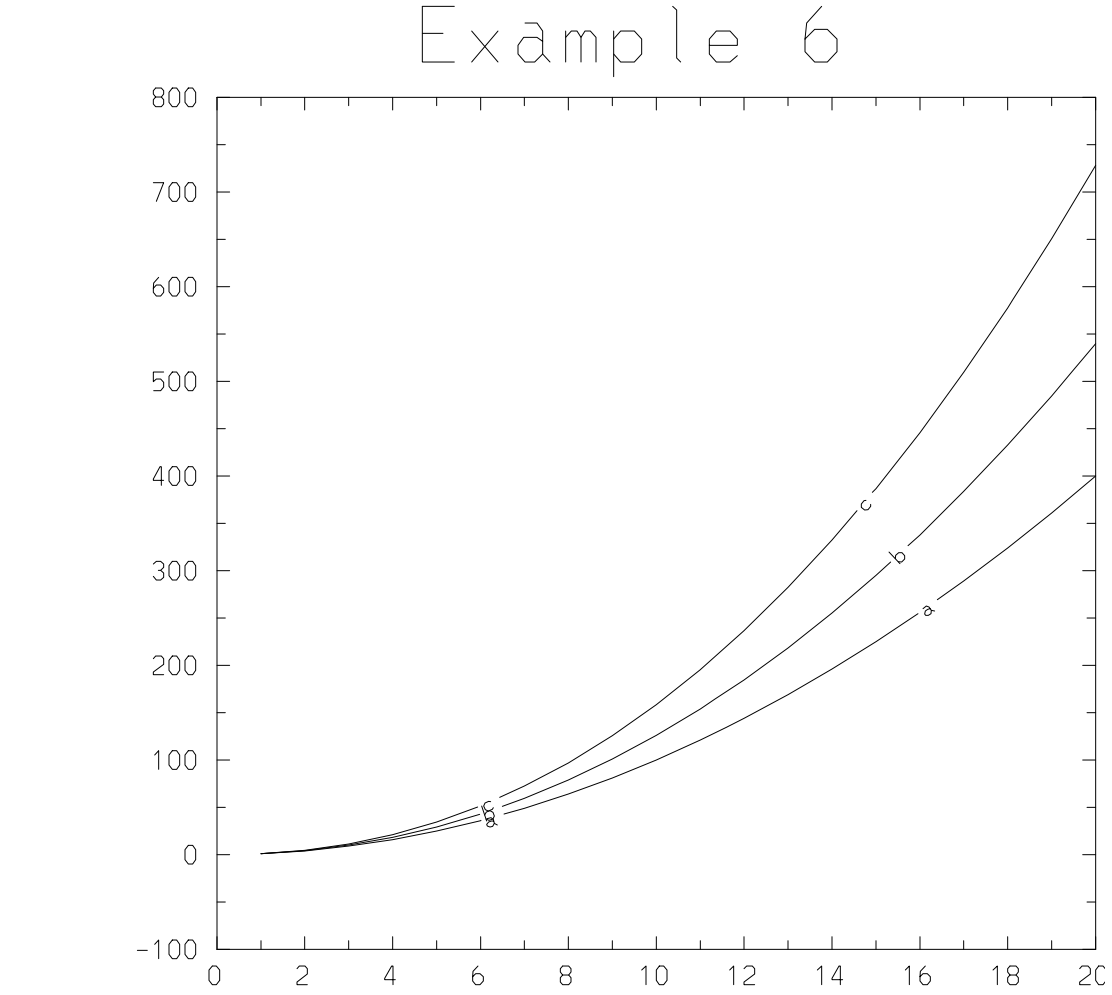


Figure 44.5: Example 5

```
# Example 6
nf
character*8 pwrs=["a","b","c"]
titles "Example 6",
   "Three Curves on One Plot, Labeled (Vector Syntax)"
plot [y1,y2,y3] x color=red labels=pwrs
sf
```




```
          Three Curves on One Plot, Labeled (Vector Syntax)
a-c: plot [y1,y2,y3] x color=red labels=pwrs
```

Figure 44.6: Example 6

```
# Example 7
nf
titles "Example 7","Three Curves on One Plot, Markers"
plot y1 x color=red mark=circle
plot y2 x color=blue mark=plus
plot y3 x color=green mark=cross marksize=2.
sf
```

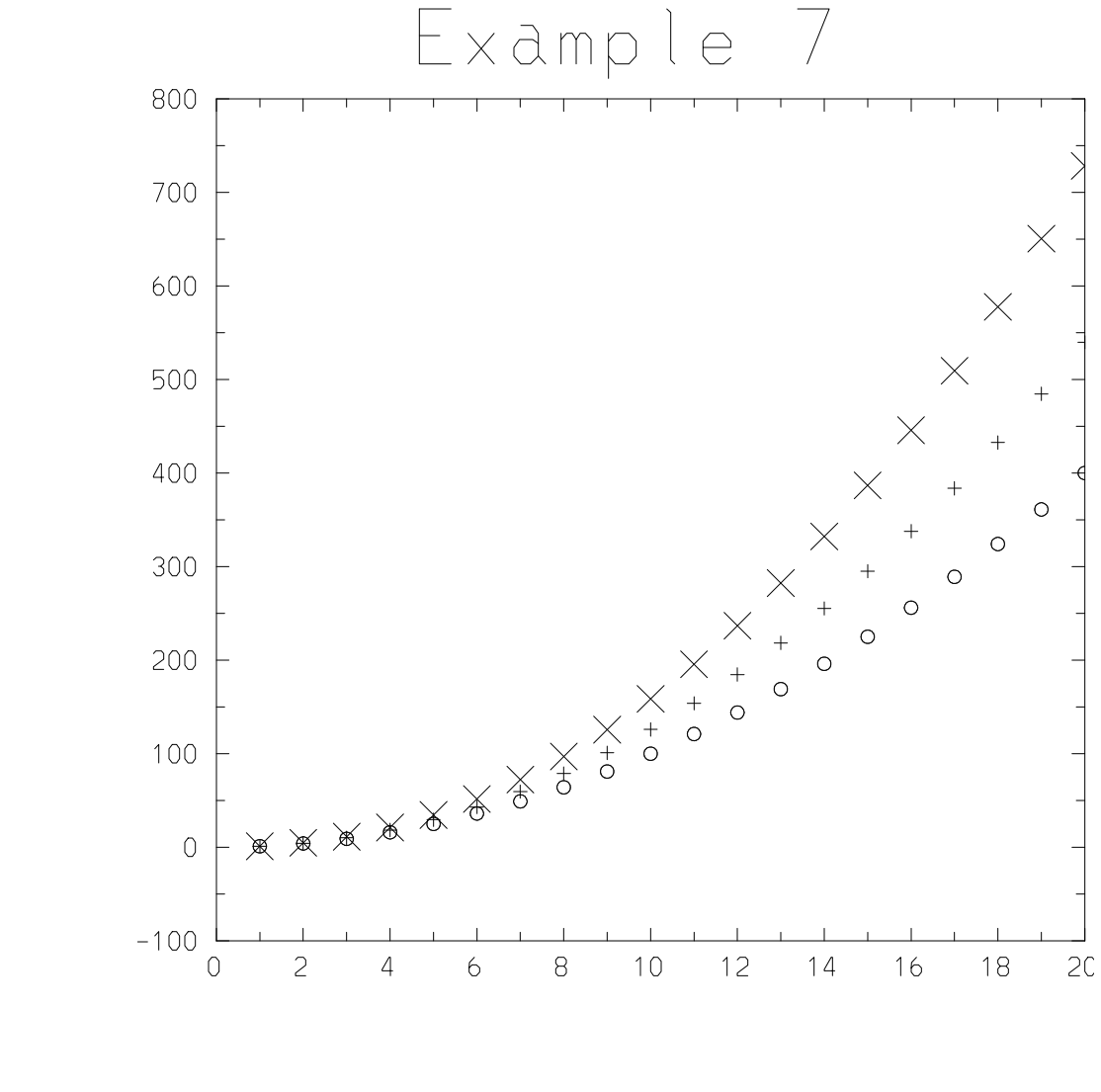


Figure 44.7: Example 7

```
# Example 8
nf
ezctitle="Supertitle appears on all subsequent frames"
titles "Example 8","Log(X)","Log(Y)"
plot [y1,y2,y3] x color=rainbow scale=loglog
sf
```

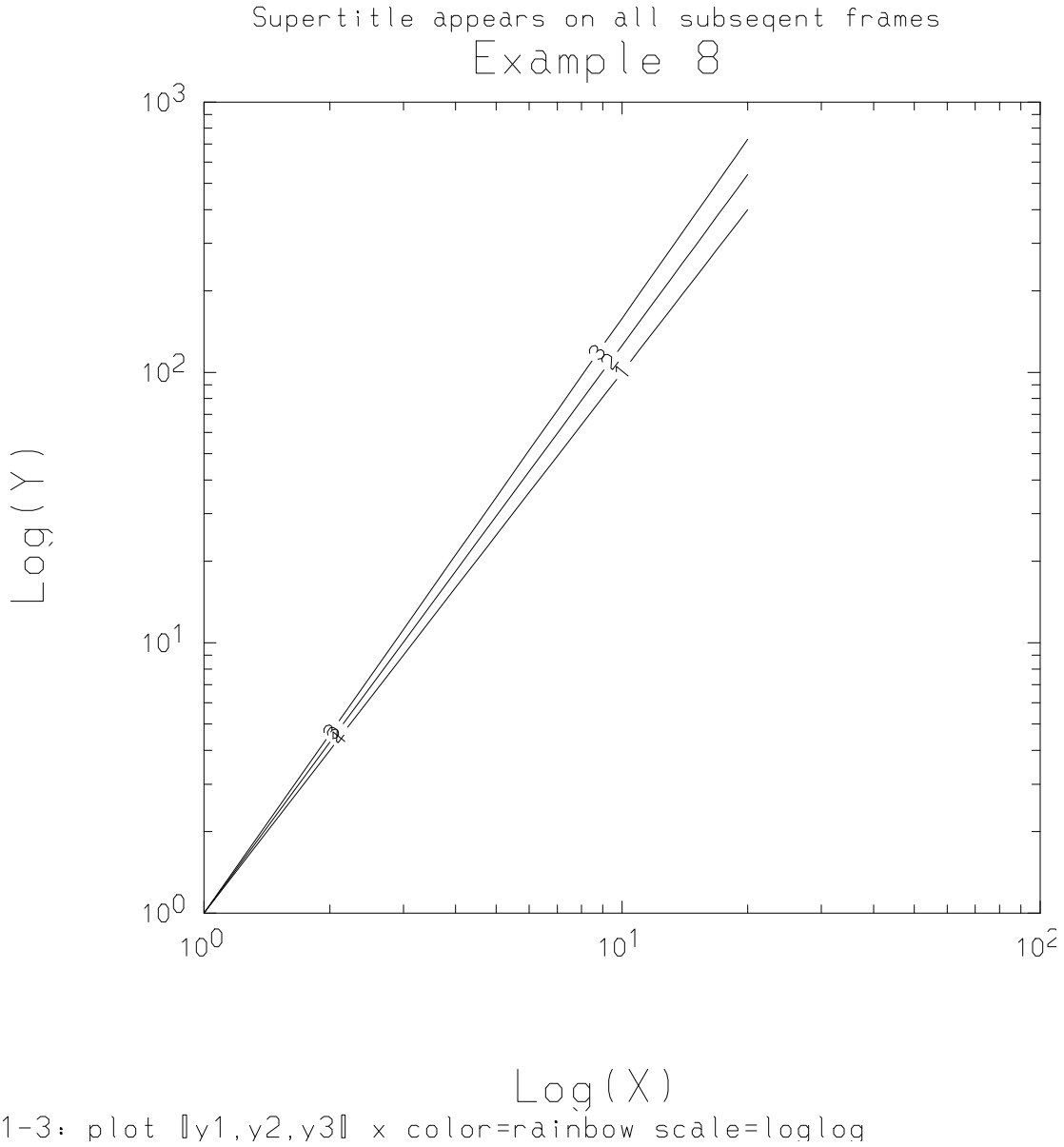


Figure 44.8: Example 8

```
# Example 9
nf
titles "Example 9","Contour Plot"
real x=iota(-5,5)
real y=x+6
real z=outer(x,y)
plotz z x y color=green lev=12
sf
end
```

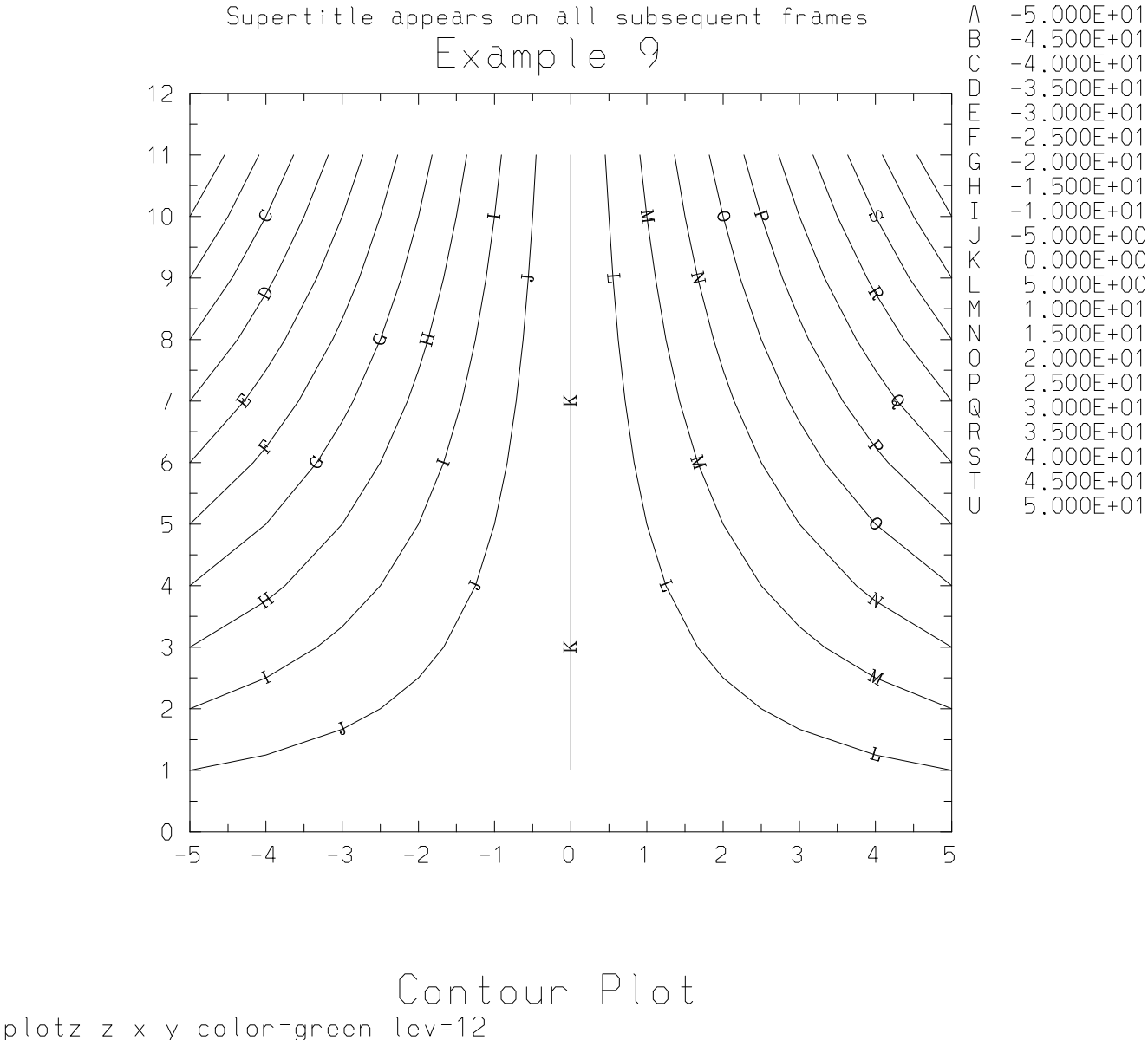


Figure 44.9: Example 9

44.2 Incorporating EZN in your program

Authors using the standard Basis makefile-creating program `mmm` will have EZN graphics by default. The `-nog` option to `mmm` will make the program without graphics. See the `mmm` man page for details. If you wish to make your own makefiles, read on.

Authors may add EZN to their program by including the file ezn.pack, found in the `include` subdirectory of the Basis installation (usually `/usr/local/basis/include`), in the input to the `config` program.

Load with the binary file pkgezn.o (found in the `lib` subdirectory of the Basis installation). You will also need to load with the appropriate NCAR library for your site. Basis comes with a utility `mmm` which can make the correct Makefiles for each site and architecture. If you wish to use your own Makefile system, the source for `mmm` can be used to deduce the appropriate information.

CHAPTER

FORTYFIVE

Devices

The EZN package has commands to control graphics devices. The devices supported by EZN are NCAR CGM files, PostScript(PS) files, and Xwindows. With NCAR3.2 or later distribution, NCAR-GKS supports Xwindows. With NCAR4.0 or later, PS output is available and multiple windows are fully supported. The current version of the package requires NCAR4.0.1 or later and (in Basis 12.0 or later) supports only NCAR GKS.

A user can open multiple devices and direct the graphics output to different devices. For example, a user can open several Xwindows, even at different workstations, and display different frames in different windows for comparison. When the user is satisfied with the result of a certain frame, he/she can issue `cgm send` to record the frame into a CGM file, or `ps send` to record it to a PS file.

45.1 Device Commands

The device commands are of the form:

device-type device-command command-modifier

Here *device-type* can be: `cgm`, `ps`, or `win`. *device-command* can be: `on`, `off`, `close`, `send`, `list`, `slist`, or `colormap`. *command-modifier* can be: `mono`, `color`, or a string for *window-name*.

The device **cgm** is a CGM file. The CGM file stores the frames of graphics output. The file produced, called an NCGM file, has a special format that NCAR utilities use. NCGM files have suffix `.ncgm`. 45.2See 3.2 “CGM File Output” on page 17 for more details. A CGM logfile, with suffix `.cgmlog`, is also created to record the frame numbers and each command issued in the frame.

The device **ps** is a PostScript file, which has suffix `.ps`. The PS file stores the frames of graphics in the PostScript format. A PS logfile similar to the CGM logfile is created with suffix `.pslog`.

The device **win** (or **tv**) is an Xwindow on a certain “display”. The “display” is the network address set by the user’s environment variable DISPLAY. The variable ezcdisp can be set by the user to redirect the window display.

The command "on" opens a device if the device has not been opened. Then it activates the device. It has no effect on the device if it is currently active. "open" is a synonym for "on".

The command "off" deactivates an opened device (but the linkage to the device for controlling still exists). The command "close" deactivates and then closes the device.

The command "send" sends the current frame to the specified device. If the target device is a CGM or PS file, the send command turns on the device (i.e. CGM or PS file), sends a frame, and then turns the file off. If the target device is an X window, then the current frame is re-sent to the device provided the target device is active.

The commands "list" and "slist" apply only to device **win**. For descriptions, 45.3see Section 3.3 "Working with Windows" on page 18.

For information about the command "colormap", 45.5see Section 3.5 "Setting the Colormap" on page 20.

The *command-modifier* is used to specify additional properties of the device. All devices default to color. When the user wants to override this default, he/she can supply the modifier mono when the device is opened the first time. (Modifier color is provided for compatibility with earlier versions in which mono was the default for device **ps**.)

The modifier *window-name* is used to label the window in order to identify it for future commands. The window name appears in the title bar of the window just opened. For more details, 45.3see Section 3.3 "Working with Windows" on page 18.

EZN keeps track of the number of active devices. If a plot command is issued without any active device, EZN will open a CGM file to accept the plot command.

Example 1

This example illustrates the use of the "open" and "send" commands.

```
win on
    # Open an Xwindow without name.
plot iota(20)**0.5
plot iota(20)**1.2
cgm send
    # Open CGM file, send a frame to it with two curves,
    # then immediately set CGM file to off.
nf
plot iota(15)**1.2
    # The plot appears on the window.
cgm send
    # Activate the CGM file to accept a frame,
    # then deactivate the CGM file.


ps send
    # Open PS file, send a frame to it,
```

```
    # then deactivate the PS file.
nf
plot iota(20)**1.5
    # The plot appears on the window.
ps send
    # Re-activate PS file, send a frame,
    # then deactivate the PS file.
end
    # Close all devices; close CGM file and PS file.
    # Examine logfiles to see what has been sent to each.
```

45.2 CGM File Output

Recall that the NCAR form of CGM file is different from the standard CGM file. Utilities for processing these files work on one format or the other.

The NCAR utilities `cgm2ncgm` and `ncgm2cgm` convert from one form to the other:

```
cgm2ncgm < foo.cgm > nfoo.ncgm
ncgm2cgm < nbar.ncgm > bar.cgm
```

The `gist` utility developed at LLNL can be used to view either a standard or NCAR style CGM file interactively. You issue the command `gist` (or `agist`) to invoke the tool. Type "`gist -h`" for a short usage summary or see its man page for details.

The NCAR utilities `idt`, `ictrans`, and `ctrans` (found in `$NCARG_ROOT/bin`) can be used to view and print NCGM files. See their man pages for details.

The `idt` utility of NCAR can be used to view a NCGM file interactively. You issue the command `idt` to invoke the tool.

Here is how to print all the frames in an NCGM file on a monochrome laser printer:

```
ctrans -d ps.mono foo.ncgm | lpr
```

Here is how to view the frames in an NCGM file on a Tektronix 4010 terminal:

```
ctrans -d t4010 foo.ncgm
```

For a complete list of the devices supported by your local NCAR distribution, execute NCAR utility `gcaps` (found in `$NCARG_ROOT/bin`).

45.3 Working with Windows

In order to display graphics to an Xwindow it is necessary to first open it. By default, "`win on`" creates a window seven inches square. One may set variables `ezcwinht` and/or `ezcwinwd` to different sizes (in inches) before opening the window to override this default.

If multiple windows will be used, then window names must be provided on the `win` command to identify the windows for activation or closing. Only one window is active at any time for a given "display". The user can apply this feature to display different plots in different windows for comparison purposes.

Note that if a second window is opened, it will most likely appear on top if the first and will need to be repositioned to make both visible on the display. Study Example 2, below, for more information on working with multiple windows.

If there are only two windows connected, a "`win close`" to one of them automatically activates the other. On the other hand, if more than two windows are connected and the active window is closed, one must explicitly do a "`win on`" to activate the desired window.

To get a list of the currently open windows, type "`win list`". This will display at the terminal a list of the currently open windows in the order created, with their status `st` (T for active, F for inactive), workstation id `wsid`, connection id `wsconn`, and the associated display device. The exact format of the `wsconn` field is platform-dependent. For a short list containing only the first three columns type "`win slist`". Both forms are illustrated in the following example.

Example 2

Try out the following commands to gain experience working with multiple devices.

```
tv on window1        # (Same as "win on window1")
    # Create a window named "window1".
plot iota(20)**2 color=red
win on window2
    # Create another window named "window2";
    # the first window is deactivated.
plot iota(20)**1.5 color=green
    # The plot will appear on window2.  Note: both curves
    # appear, since EZN maintains a single display list.
nf
    # Set a new frame.
win list
    # List currently open windows.

 idx st  name              wsid     wsconn (hex)    display
 ------------------------------------------------------------
   1  F  window1           1001   240000100000000  128.115.36.49:0.0
   2  T  window2           1002   300000100000000  128.115.36.49:0.0
win on window1
```

```
    # Reactivate window1; window2 is now inactive.
win slist
    # Use short list to verify change of status.
 idx st  name
 ------------------------
   1  T  window1
   2  F  window2
plot iota(15)**2 color=blue
    # The plot will appear on window1, superimposed on what
    # was there already, because "nf" applies only to active
    # window.  Need a new "nf" after the "win on" to clear.
...
win close window1
    # Close window1; now window2 is automatically active.
nf
plot iota(10) color=magenta
win close window2
    # Close window2; now no devices are active.
nf
...


cgm on
    # Open a CGM file to accept the following plots.
    # (Note that this would happen automatically after the
    # next plot command, since no device is currently active.)
plot iota(20) color=yellow
    # A frame is send to the CGM file.
ps on
    # Open a PostScript file.
plot iota(20)**1.2
    # A frame has been send to both CGM and PS file.
nf
cgm close
    # Close CGM file.
plot iota(20)**1.8
    # A frame has been send to PS file.
ps close
    # Close PS file.
end
    # Implicitly close all active devices.
```

45.4 Setting the Background Color

The default background color is white for cgm and postscript devices, black for X-windows. The default can be overridden by executing one of the following calls *before* the device is opened:

```
call
ezcsetbw -- set the background color to white.call
ezcsetbb -- set the background color to black.
```

The foreground color will be the complementary color.

45.5 Setting the Colormap

The "`colormap`" command can be used to alter the default colormap that is installed when a color device is opened.

The command has the form

device-type `colormap` *map-name*

where *map-name* should be "`idl`*n*", 1¡=*n* ¡=16, to select one of the 16 IDL colormap that have been loaded into EZN. Some of these have alternate, more descriptive names:

Original Alternate

idl1 greyscale

idl2 bluescale

idl4 brownscale

idl7 rainbow

idl8 pinkscale

idl9 greenscale

For complete control over the colormap, the user can specify *map-name* "`mycolormap`", which causes the RGB definition of the colormap to be loaded from the arrays `ezcred`, `ezcgreen`, `ezcblue`. The user should set these arrays to integer values in the range 0-256 before the device command.

The colormap setting must be done at the time a device is initialized. For device **win**, EZN will close the currently open window and create a new one with the requested colormap if only one window is open. For other device types, an error return will occur if a device of this type is already active. An error message will also result in case of multiple windows. You can use "`win list`" to display open windows and "`win close all`" to close them all. The new colormap will apply to the window opened with the next "`win on`" command.

Example 3

The following example sets up a colormap very much like the default.

```
# Create and install a colormap.
integer num = 249 # The size of the ezc color arrays.
integer num2=num/2
ezcred = 0.:256.:num
ezcgreen(1:num2) = 0.:256.:num2
ezcgreen(num2+1:num) = 256.:0.:(num-num2)
ezcblue = 256.:0.:num
cgm close  # Must open new CGM file to install new colormap.
cgm colormap mycolormap
```

-

CHAPTER

FORTYSIX

The EZN Graphics Model

46.1 The Additive Model

The basic model of this package is that of additive graphics commands to a single frame. That is, each graphics command adds objects (curves, meshplots, etc.) to a frame. The frame is not complete until a newframe "`nf`" command is issued. The user controls whether or not to see each step in building a frame or just viewing the completed frame by setting the variable `ezcshow` to `true` or `false`.

EZN begins in interactive mode (the variable `ezcshow` is true), so that each command that changes the frame causes the whole frame to be redrawn. However, Lasnex, and most other programs using EZN, will set `ezcshow` to false when making plots so that each frame is displayed only when finished. Lasnex users in particular should note that any "snapshot" plot will put EZN into this mode. If you stop the program and want to view the plots as they are made, you must either reset `ezcshow` to true or use the showframe "`sf`" command.

Caution: When using multiple windows in interactive mode, be aware that "`nf`" clears the display list, but only clears the currently open window. If you then change windows, you will have to issue another "`nf`" command to avoid overplotting any plot already on the window.

46.2 Controlling Layout

The standard EZN picture can be described as follows. The picture is divided vertically to allow a fraction (`ezccntfr`) of the right side of the picture to be used to list contour level values. The remaining left side is divided horizontally to allow a fraction (`ezclegfr`) at the bottom to be used for the legend. The remaining upper part of the left side consists of the axes and its labels surrounded by the four titles (whose relative size is controlled by `ezctitfr`). The supertitle `ezctitle` goes either at the top or bottom of the left part of the frame, depending on the value true or false of `ezcsuper`.

The function `ezcminsz(minsz)` can be used to set the minimumtext size of numerical labels on the axes to a value between 6 and 24, inclusive. As this size increases, more room is left for the numerical labels. The default size is 12.

The space devoted to each of these components is, by default, allocated whether they are present or not. The variable `ezcfixed` can be set `false` to allow the space to be better used. This gives you as big a picture as possible. However, different pictures may allocate frame space differently, and hence no longer be directly comparable. For example, a mesh plot and a contour plot will be different sizes because the latter will use some of the space for the contour legend.

46.3 Plot Command Summary

Here is a summary of the commands which are described in the remainder of this manual.

- Attribute commands (47CHAPTER 5: “Attributes”)

```
attr keyword=value, keyword=value
    # set color, thickness, etc.
```

- General plot commands (48CHAPTER 6: “General Plot Commands”)

```
plot y x          # curves, markers
plotz z x y       # contours
ploti cind        # cell array plot
```

- Mesh-oriented commands (49CHAPTER 7: “Mesh-Oriented Commands”)

```
plotm           # plot mesh
plotb           # plot mesh boundaries
plotc expr      # plot contours of a mesh-based quantity
plotf expr      # fillmesh plot
plotv           # plot velocity field
plotr lasernum # plot laser rays for Lasnex
```

- Polygonal-mesh commands (50CHAPTER 8: “Polygonal-Mesh Commands”)

```
plotp x y         # plot polygonal mesh
plotpf expr x y   # polygonal fillmesh plot
```

- Surface plot commands (51CHAPTER 9: “Surface Plot Commands”)

```
srfplot      # wire-frame surface plot
isoplot      # wire-frame isosurface plot
```

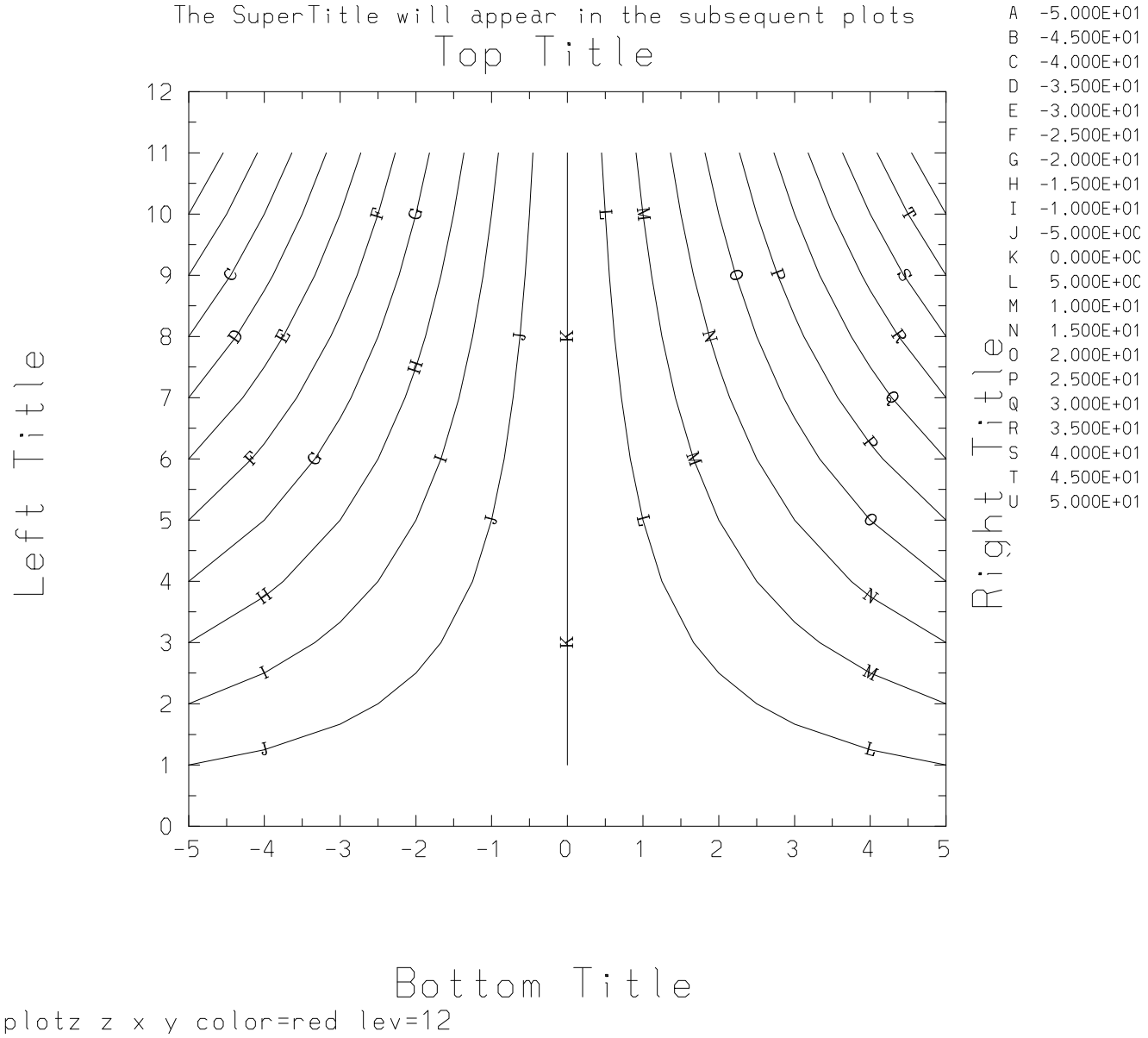


Figure 46.1: Example of frame layout

- Frame control (52CHAPTER 10: “Frame Control”)

```
frame xmin,xmax,ymin,ymax
nf           # begin new frame
sf           # display current frame
undo number  # remove number’th command in the frame
```

- Text (53CHAPTER 11: “Axes, Titles and Text”)

```
titles "top","bottom","left","right"
ezctitle = "supertitle for all frames"
text "message",x,y,charsize,angle,centering
ftext "message",x,y,charsize,angle,centering
stdplot << "X = " << x
output graphics  # direct output to the graphics device
....ordinary Basis output
output tty       # direct output to the terminal
```

- Quadrant Control (55CHAPTER 13: “Quadrant Mode”)

```
ezcquad(iquad)
ezcsquad(xmin,xmax,ymin,ymax)
```

- Interactive graphics (56CHAPTER 14: “Interactive Graphics Tools”)

```
read interactive.in
zoom / unzoom           # zoom in on subframe
markp / markpp          # mark point(s)
markl / markll          # mark line(s)
marks / markss          # mark segment(s)
markr / markrr          # mark region(s)
markz / markzz          # mark zone(s)
```

You can use attributes and the values of user-settable variables to control the detailed behavior of these commands. Attributes are explained in the next chapter, variables in 57CHAPTER 15: “Control Variables and Defaults”.

CHAPTER

FORTYSEVEN

Attributes

A set of “attributes” such as color, line thickness, scale, marks, labels, etc., can be used to control the appearance of graphics objects or the layout of a frame.

47.1 Attribute Types

Some attributes affect the entire picture (such as scale, frame limits) while others affect the individual graphic objects in the picture (such as thickness, color).

If the attribute affects the entire picture, it will take effect immediately and we call it a *frame* attribute. If the attribute only affects the individual graphic object, we call it an *object* attribute. A special kind of object attribute (for mesh plots), which affects the current object and remains in effect until a frame advance or until another assignment is made to the attribute, is called ”*sticky*”. 47.3See “Attribute Table” on page 31 for a list of valid keywords, values and their attribute types.

The `grid` and `scale` attributes are examples of *frame* attributes. These attributes affect the entire picture. When these attributes are specified with the `attr` command or on an EZN graphic command line, a new picture is plotted with the grid and scale changed. (Note: This has the side effect of creating a new frame even if the variable ezcshow = false. To avoid the generation of extra frames, then, it is necessary to issue the `attr` and `frame` commands specifying the frame attributes *before* any plot commands for the frame.)

The `color` and `style` attributes are examples of *object* attributes. If these attributes are specified on a graphic command line, the color and line style are changed only for the objects generated by this command. If these attributes are specified with the `attr` command, only those objects added to the frame following the `attr` command will have these specified attributes. Some special attributes for the mesh plots such as region, krange, lrange are ”*sticky*”, i.e. the specifications of region, krange and/or lrange will affect the following mesh plots until the end of the frame or the values have been redefined.

If no attribute value is set explicitly by the user, a default value will be used for the attribute. These default values in turn can be changed by setting certain control variables. User specified default values will be in effect until new default values are assigned. For details, 57See CHAPTER 15: “Control Variables and Defaults” on page 101.

By specifying attributes and control variables, it is also possible to change many things about the layout of the picture, such as the portion of the picture used for the legend, the portion of the picture used for the contour level annotations, the size of the titles, and the minimum size of the text.

Usually all attributes will be re-initialized to their default values when a frame is advanced. However, setting the variable ezcreset to `false` will cause the attribute settings to last across frames.

Examples

```
plot y1,x1
plot y2,x2
attr scale=linlog       # Picture redisplayed.
nf
plot y1,x1
plot y2,x2
attr style=dashed       # Only following curves affected;
        #        no redisplay yet.
plot y3,x3
plot y4,x4
nf
```

The attributes `legend`, `labels`, and `lev` can be either *frame* or *object* attributes. For example, `legend` can be set to `yes` or `no`, to indicate whether or not the graphic commands are to be listed at the bottom of the frame, thus supplying a handy index for each object generated by the graphic command on the frame. As an *object* attribute, `legend` can also be set to any arbitrary string for a particular command by specifying the `legend` attribute with the command, as in:

```
plot y x legend="Pressure versus density"
```

This results in the string "`Pressure versus density`" being listed as the command at the bottom, rather than "`plot y x`" which would result if this option were omitted.

The legend attribute used as an *object* attribute can also be used to suppress the legend for the current object, as in:

```
plot y x legend=" "
```

This causes the current command not be listed in the legend list.

Labels for the curves can be specified with the `labels` keyword. Labels must be quoted strings, or variables or expressions (including arrays) whose values are quoted strings. The attribute `labels` is also used to turn labelling on and off (by setting it to `yes` or `no`). When the attribute `labels` is used in this sense, it is a *frame* attribute. i.e., all existing and subsequent curves on the frame will be either labelled or not.

The attribute lev can be used to assign the number of contour levels or a vector of contour level values as an object attribute. When lev=log, it becomes a *frame* attribute, it sets the contour plots based on logarithmic scale.

47.2 attr: Setting Attributes

Calling Sequence

```
attr
keyword1=value1, keyword2=value2, ... , keywordN=valueN
```

Description

The **attr** command assigns values to attributes. These keyword=value pairs can be either comma or space delimited. The value assigned to an attribute remains in effect until a frame advance is issued, or until another assignment is made to the attribute via the `attr` command (within the same frame). To make the values assigned to attributes remain in effect across frame advances, set variable `ezcreset` to `false`.

To make a permanent change to a default, change the corresponding variable. For a list of these, 57See CHAPTER 15: “Control Variables and Defaults” on page 101.

Examples

In the first example, the scale is set to `loglog`, the line style is set to `dashed`. Since the default value for variable ezcreset was used, the attributes set only remained in effect until the next frame advance. After that, the attributes are reset to their default values.

```
# ezcreset=true (default)
# Settings remain in effect only until next frame advance.
attr scale=loglog,style=dashed
plot y1,x1
plot y2,x2
nf
plot y3,x3          # scale,style reset to defaults.
```

In the second example, variable ezcreset is set to false. This time the `attr` command remains in effect across frame advances. Hence, the line thickness remains set to 1.2 across frame advances.

```
ezcreset=false
# Settings remain in effect across frame advances.
attr thick=1.2
plot y1,x1
plot y2,x2
nf
plot y3,x3          # Thickness still 1.2.
```

Or, we could accomplish the same thing more simply by making a permanent change to the default thickness:

```
# ezcreset=true (default)
defthick=1.2
plot y1,x1
plot y2,x2
nf
plot y3,x3          # Thickness still 1.2.
```

47.3 Attribute Table

The following is an alphabetical list of all allowable attribute keywords. Refer to individual plot commands for more specific information.

Keyword	Type	Value	Description
arrow	object	no	No arrows on curve (default).
		yes	Plot arrows on curve.
bnd	object	no	Plot full mesh (default).
		yes	Plot region boundaries only.
color	object	bgcolor,fgcolor	The default background / foreground color used by EZN.
		color	Use one of the following 16 named colors (default=fgcolor): red, green, blue, cyan, magenta, yellow, coral, yellowgreen, springgreen, slateblue, skyblue, orangered, gray33, lavender, orchid, gray70.
		rainbow	Colors run down through the list of named colors. (*See* note after table for more details.)
		filled	Color fill the contour band, ranging from blue to red.
		rfill	Color fill the contour band, ranging from red to blue.
		fillnl	”filled” without contour lines.
		rfillnl	”rfill” without contour lines.
		power	Ray color varies along path to show intensity. (*See* 49.57.5 “plotr: Lasnex Rayplots”.)
		relpow	Ray color shows relative intensity. (*See* 49.57.5 “plotr: Lasnex Rayplots”.)
cscale	object	lin	Use linear color mapping (default). (*See* 49.37.3 “plotf: Fillmesh Plot”.)

Keyword	Type	Value	Description
		log	Use logarithmic color mapping.(*See* 49.37.3 “plotf: Fillmesh Plot”.)
		normal	Use color mapping based on the normal distribution. (*See* 49.37.3 “plotf: Fillmesh Plot”.)
		rlin	“lin” with reversed color order.
		rlog	“log” with reversed color order.
		rnormal	“normal” with reversed colors.
grid	frame	no	No reference grid
		tickonly	Tick marks only (default)
		x	x rulings
		y	y rulings
		xy	x and y rulings
kcolor	object	color	Use *color* for k-lines. (*See* color; default: current color attribute)
krange	sticky	*kmin:kmax:kinc*	Range for k-lines in mesh plot.(default=1:kmax:1)
kstyle	object	none	No lines in k direction.
		style	Use *style* for k-lines. (*See* style; default=solid)
labels	frame	yes	Curves/marks are labelled in the order added (default).
		no	No labels displayed.
	object	*str*	Label next curve with *str*. *str* can be a vector for multiple curves.
lcolor	object	*color*	Use *color* for l-lines. (*See* color; default: current color attribute)
legend	object	*str*	User-specified legend in quotes (maximum 120 characters). By default, the command line is used.
	frame	yes	Legend plotted below the frame (default).
		no	No legend plotted.
lev	object	*ival*	Number of levels (default=8). (For complete specification, *see 48.2.1*6.2.1 “Contour Levels”.*)*
		[rval]	Vector of contour levels.
	frame	linear	Linear contours (default).
		log	Logarithmic contours.
lrange	sticky	*lmin:lmax:linc*	Range for l-lines in mesh plot. (default=1:lmax:1)
lstyle	object	none	No lines in l direction.

Keyword	Type	Value	Description
		style	Use *style* for l-lines. (*See* style; default=solid)
mark	object	asterisk	Use asterisk marker.
		circle	Use circle marker.
		cross	Use cross marker.
		dot	Use dot marker.
		plus	Use plus marker.
		x	Use x marker.
marksize	object	*rval*	Scaling factor for markers (default=1.).
point	object	no	Physics quantity is zone-centered (default). (*See* 49CHAPTER 7: "Mesh-Oriented Commands".)
		yes	Physics quantity is defined at mesh points.
region	sticky	all	Display all regions in mesh plots (default).
		[ival]	Vector of desired region numbers.
rsquared	object	*rval*	Multiquadric r-squared parameter.
		0.	Program calculates (default).
scale	frame	linlin	Both x and y axes linear (default)
		linlog	x-axis linear, y-axis logarithmic
		loglin	x-axis logarithmic, y-axis linear
		loglog	Both x and y axes logarithmic
		equal	Both x and y axes linear, scales equalized
style	object	solid	Solid lines (default)
		dashed	Dashed lines
		dotted	Dotted lines
		dotdash	Dot-dashed lines
		ltor	Mark curve with arrows pointing left-to-right.
		rtol	Mark curve with arrows pointing right-to-left.
		pm	Plus/minus (for contour plot)
		none	Background color (invisible) lines
thick	object	*rval*	Line thickness multiplier(default=1.)
vsc	object	*rval*	Vector scaling factor (default=0.05).
zlim	object	[*zmin*,*zmax*]	Limits to be used in defining colormap for fillmesh plot (default=limits of plotted array).

Note: In the `color` array that assigns names to colors, ”bgcolor” has index 0, ”fgcolor” index 1. The named colors are numbered from 2 to 17. ”color=rainbow” cycles through indices 1 through 13 only. When applied to a vector of curves, the cycle starts at 2 (red); for contour lines, the cycle starts at 4 (blue).

CHAPTER

FORTYEIGHT

General Plot Commands

This chapter describes the EZN general-purpose plot commands.

48.1 plot: Plotting Curves and Markers

Calling Sequence

```
plot <yexpr>, <xexpr>, <keylist>
```

Description

The **plot** command plots line segments connecting points or discrete markers at the points. Markers are plotted at the data points, without connecting line segments, when the attribute `mark` is set to one of the valid marker types. (An invalid marker type is treated as `mark=x`.)

The default scaling factor for markers is 1.0, the default line style is `solid`, and the default line thickness is 1.0. To override these values, set the attributes `marksize`, `style`, or `thick`, respectively. (Due to the Xwindows "clip" characteristics of NCAR, markers that would be partially drawn beyond the frame limits are completely clipped. In order to see the markers at the frame limits, the user needs to use a frame command to extend the limits to include the whole markers.)

By default, the curve is plotted with both axes on a linear scale. For logarithmic plots, set attribute `scale` to "`linlog`", "`loglin`" or "`loglog`".

If neither *yexpr* nor *xexpr* is specified, the current picture is redisplayed. Otherwise, *yexpr* is an array of y-axis values, *xexpr* is an array of x-axis values, and ¡*keylist*> is a list of optional attributes specified by pairs of keywords and values.

If *xexpr* is not specified, then *yexpr* is plotted against the previous version of *xexpr*. If this value does not exist, then *yexpr* is plotted against the index of *yexpr*. If plotting against the index of *yexpr*, the lower and upper subscripts of *yexpr* are used as the starting and ending points, respectively.

If *yexpr* differs in length by one from the length of *xexpr*, whether explicitly or implicitly specified, the longer of the two may be automatically averaged to shorten it. Set variable ezcnocx or ezcnocy to `false` to disable averaging. If averaging is not permitted, the command is an error and no object is added to the frame.

If the arguments are two-dimensional arrays, `plot` plots the corresponding columns of *yexpr* and *xexpr* to produce multiple curves at once. Multi-dimensional arguments are reduced to two-dimensional by collapsing any higher dimensions. If *xexpr* is one-dimensional, then each column of *yexpr* is plotted against it.

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified; i.e., they are not remembered across commands.

`grid`, `scale`, `style`, `thick`, `color`, `arrow`, `labels`, `font`, `mark`, `marksize`, `legend`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

Although it is recognized, the `font` attribute currently has no effect.

Examples

In this example, three curves will be superimposed. The first plot command will plot a curve with dashed lines, the second plot command will mark circles twice the default size, and the third plot command will plot the curve in red. Since the first plot command does not specify *xexpr*, y will be plotted against an array spanning from 5 to 15. In the second and third plot commands, the y values are plotted against the previous expression of x. The curves are labelled 1 and 3 respectively. (“Marked” plots are not labelled.)

```
real y=iota(5,15)
plot y,style=dashed                # plot curve
plot y+1,mark=circle,marksize=2    # plot markers
plot y+2,color=red                 # plot curve in red
nf
```

If you enter the above commands at the terminal, you will see three frames displayed in turn as the graphic objects are built up, and the `nf` command will clear the screen. If you repeat the experiment with `ezcshow=false`, you will not see any graphic objects at all until the `nf` command, at which point the completed frame will appear.

The next example replots two curves with an xy-grid added.

```
real x=0.5*iota(1,10)
real y=x**2
plot y,x
plot y-1,x
plot grid=xy
nf
```

In the next example, the first plot command will plot three curves, y,y+1,y+2 against the same x, labelled “a”, “b”, and “c”, respectively. The second plot command will plot two curves, y+3 against x+1 and y+4 against x+2, labelled 4 and 5, respectively. (Note that the curve number continues to increment even if the number is not the curve label.)

```
plot [y,y+1,y+2],x,labels=["a","b","c"]
plot [y+3,y+4],[x+1,x+2]
nf
```

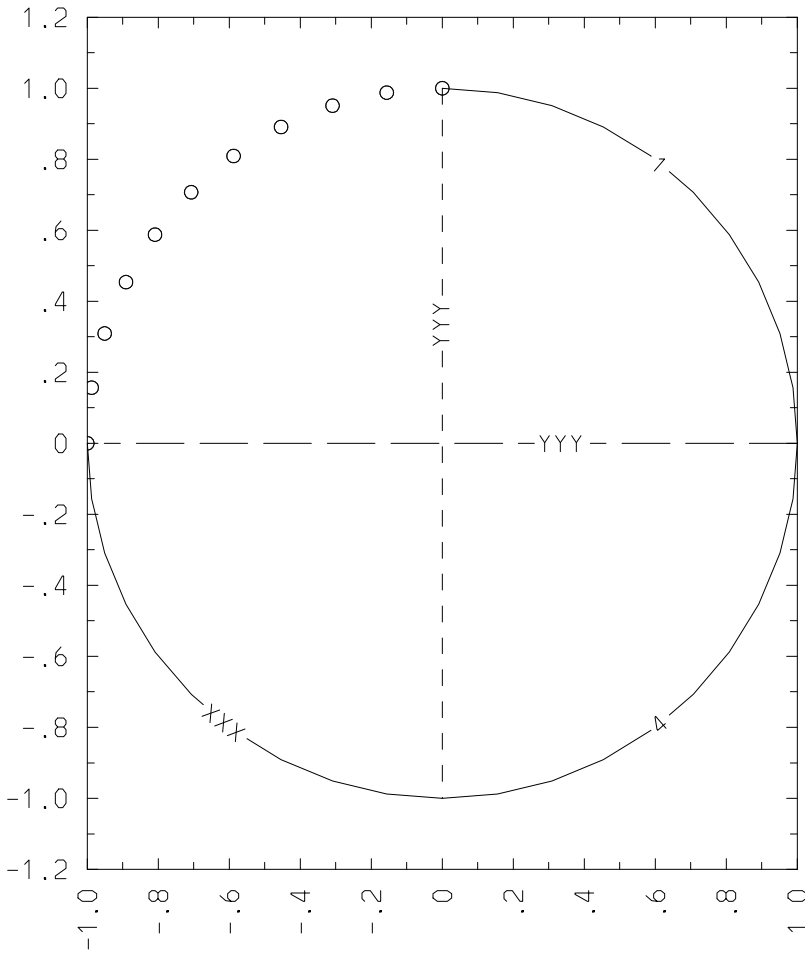
The fourth example shows how to set the legend.

```
plot y,x,legend="this is my legend"
nf
```

The fifth set of examples graphs the unit circle and x and y axes in a variety of styles and also illustrates how the `labels` attribute works. Comments in the code explain what happens on the frame.

```
attr scale=equal
# Set x and y scales be equal:
$a=(pi/2.)*iota(0,10)/10.
# Curve in first quadrant labelled with 1:
plot cos($a) sin($a) legend = "first quadrant"
# Curve in second quadrant not labelled "Q2" since
#   drawn with a "mark":
plot cos($a) -sin($a) labels ="Q2" mark=circle
# Third quadrant drawn and all labels turned off, but
#   label "XXX" is still associated with quadrant 3:
plot -cos($a) -sin($a) labels=no labels="XXX"
# All labels turned back on, including "XXX" in quadrant 3;
#   quadrant 4 labelled with 4:
plot -cos($a) sin($a) labels=yes
attr labels="YYY"
# The following two curves will now be labelled with "YYY":
plot 0*ones(11) 1.//((5.-iota(10))/5.) style = dashed
plot 1.//((5.-iota(10))/5.) 0*ones(11) style = dotted
```

See the following figure for the completed frame. (Note that the figure was generated before NCAR graphics totally clipped markers that are partially beyond the frame limits, so the picture you get by executing these commands will differ somewhat from what you see here.)




```
1: first quadrant
plot cos($a) -sin($a) labels="Q2" mark=circle
XXX: plot -cos($a) -sin($a) labels=no labels="XXX"
4: plot -cos($a) sin($a) labels=yes
YYY: plot 0*ones(11) 1.//((5.-iota(10))/5.) style=dashed
YYY: plot 1.//((5.-iota(10))/5.) 0*ones(11) style=dotted
```

Figure 48.1: Example of labelling and legend specification

48.2 plotz: Plotting Contours

Calling Sequence

`plotz <fexpr>,<xexpr>,<yexpr>,<keylist>`

Description

The **plotz** command plots contours of a surface defined by *fexpr* above the point set described by *xexpr* and *yexpr*. ¡*keylist*> is a list of optional keywords and values.

There are three allowed types of data for contour plots:

- Gridded data: *xexpr* and *yexpr* are one-dimensional arrays, say x and y, and *fexpr* is a two-dimensional array, say z, such that z(i,j)=f(x(i),y (j)), i=1,...,length(x), j=1,...,length(y). In order for *xexpr* and *yexpr* to form a valid rectangular grid, each array must contain either strictly increasing or strictly decreasing values.
- Mesh data: *fexpr*, *xexpr* and *yexpr* are all two-dimensional arrays of the same shape. In this case, *xexpr* and *yexpr* form a logically rectangular mesh and *fexpr*(i,j) is the value associated with point (*xexpr*(i,j),*yexpr*(i,j)). For mesh-based data, a plot of this type can also be generated by the `plotc` command; 49.2see Section 7.2 ”plotc: Plotting Contours” on page 54.
- Scattered data: *fexpr*, *xexpr* and *yexpr* are all one-dimensional arrays of the same length. In this case, a rectangular mesh containing the data is created and *fexpr* is interpolated to this mesh by the MultiQuadric (MQ) method. This is the only case in which the optional attribute `rsquared` is used.

Note: *fexpr* can also be the name of a function or macro which, when called with no arguments, returns an array of values of the appropriate shape.

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified; i.e., they are not remembered across commands.

`grid`, `scale`, `thick`, `style`, `font`, `mark`, `marksize`, `lev`, `color`, `rsquared`, `legend`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

The default line style is `solid` and the default line thickness is 1.0. The default color is the foreground color. To override these defaults, set attributes `style`, `thick`, `color`, respectively. The `mark` attribute will cause markers to be plotted at each of the mesh points, in the foreground color.

Although it is recognized, the `font` attribute currently has no effect.

48.2.1 Contour Levels

Contour levels are controlled by the `lev` attribute. The attribute `lev` can be used to specify the levels of contours, the scale of the contours (*linear* or *logarithmic*), or a list of specific values for the contour levels. The attribute `lev` can be set either on a plot command or with an attribute command such as "`attr lev=foo`". Like any such attribute, if set with `attr` it applies to all `plotz` commands on that frame, except those that override it with a "`lev=` " of their own. However, if a vector of values is specified for `lev`, it will be lost at the next frame advance. There is currently no way to specify such a list to be used on all frames.

In "`lev=foo`", `foo` can be:

- `linear`: at least abs(deflev) linear levels;
- `log`: abs(deflev) logarithmic levels;
- *n>0*: at least *n* linear levels;
- *n¡0*: abs(*n*) logarithmic levels;
- a real or double precision list of values. (These values must be in increasing order.)

The contour plot is generated by the NCAR Conpack package. When NCAR chooses linear levels, it chooses at least the number specified but may choose up to 2**n* levels.

The default value of `lev` is in the variable deflev, whose value is 8; hence, the default is at least 8 linearly-spaced contour levels.

Every contour line is labeled. The variable ezclabel can be set to "`alpha`", "`on`", or "`off`". This will result in contours which are labeled with single letters, with contour level values, or with nothing, respectively. The default is ezclabel=`alpha`.

The special "`style=pm`" for a contour plot invokes the mode where positive contour lines are plotted using solid lines, and negative contour lines are dashed. (Mnemonic: `pm` means `plus/minus`.)

48.2.2 Contour Control Parameters

After a contour plot has been displayed, a set of variables is available to review the information about the set of contour levels used by NCAR. Do "`list Contours`" to see this list. Do "`list Random_Contour_Plots`" for variables related to the MultiQuadric interpolated values in the scattered data case.

For more detailed control of NCAR Conpack, the user may use three routines `cpsetr`, `cpseti`, and `cpsetc` to set real, integer and character parameters respectively. The following parameters in Conpack are set by EZN; the user should not change them:

CLS - contour level selection NCL - number of contour levels CLV - contour level CLU - contour level usage CLD - contour level dash pattern CLL - contour level width LLT - level label text SET

- calling of set by Conpack SPV - special value ORV - out of range value ILT - info level text SFS - scale factor selector

Conpack parameters whose values EZN sets just once during its initialization are:

`CWM - character width multiplier`

The default is 1.2. Set this bigger or smaller to control the size of the contour labels.

`HLT - hi/lo text labels`

The default is `"H'L"`. Set to blank via command `cpsetc("HLT"," ")` to remove the `H` and `L` labels.

`PC1 - real contour label positioning parameter`

Extremely short contours are not labelled. This parameter governs how short a contour can have a label.

Conpack parameters that the user might experiment with are:

`DPV -`

This integer controls the distance between the labels on a contour line.

The default corresponds to 3.

LLP - This controls the contour label positioning.

The inquiry routines cpgeti, cpgetr and cpgetc may used to find out the current setting of those parameters. The user should refer to the NCAR document “CONPACK, A Contouring Package” (available on the web at http://ngwww.ucar.edu/ngdoc/ng/supplements/conpack/) for detailed information.

48.2.3 Contour Color Fill

The color attribute for a contour plot can be used to generate color filled contour bands. This is an application of the “area” concept of NCAR. Each contour band is a closed polygon (coupled with frame boundaries if necessary) which can be filled with color. The user can set color= filled to fill the contour levels with colors ranging from *blue* to *red* with *increasing* altitude. Setting color=rfill will fill with colors ranging from *red* to *blue*. When color fill is applied, the contour lines may become unnecessary. The user may specify color=fillnl or color= rfillnl to avoid the contour lines being drawn.

For the advanced user, a different range of colors can be assigned when the default colormap is changed. 45.5See “Setting the Colormap” on page 20 for more information.

48.2.4 Contour Level Annotations

For the contour plots, the contour level annotations can be shown in the right margin of the frame under user's control. The value of control variable ezccntfr is the fraction of the whole frame on the right allocated to display this information. It lists the labels and their corresponding contour level values. (If labelling is suppressed, ezclabel=off, then only the contour level values will be in the annotation.)

The variable ezcconkey is used to control the appearance of the contour level annotation. Setting ezcconkey=off will cause the frame not to display the contour level annotation. (However the portion of the frame for contour level annotation is still allocated. To utilize the whole frame without contour level annotation, set ezcfixed=false and ezccntfr=0.) The default is ezcconkey=on.

The contour level annotation is color coded for easy association with the contour lines. The color assigned is the color of the contour level. There are several control variables that may be used to customize this contour level annotation.

The variable ezccksfill specifies either *solid* color fill or *hollow* fill (i.e. just color the border) for each cell containing the numerical annotation. ezccksfill="solid" specifies the solid fill; any other value, e.g. ezccksfill="hollow" (the default), will make a hollow fill. In a rare case, there may be two contour plots in the same frame. If conflicting colors are assigned to the same contour level, the contour level annotation will be hollow filled with a white border.

The variable `ezcconord` controls the order of the values in the annotation. By default, the values are in increasing order as you read down the list (and the labels, if present, are in alphabetical order). You may set `ezcconord=''decr''` to specify that the values be given in decreasing order. This may be especially useful if `ezclabel=''off''` and you are using one of the filled color options.

Example

The following example plots a matrix z versus vectors x and y. (Repeated from Example 9 in 44CHAPTER 2: "Introduction to EZN".)

48.3 ploti: Cell Array Plots

Calling Sequence

`ploti` *cell-indices*[,*xmin*,*xmax*,*ymin*,*ymax*],¡*keylist*>

Description

The **ploti** command is used to plot cell arrays in Basis. This is an application of the "area" concept of NCAR. The argument *cell-indices* is a two-dimensional array of color cell indices, which can be generated using the vector-to-color conversion functions described below. ¡*keylist*> is a list of optional keywords and values.

The user translates a two-dimensional array of physics quantities to a two-dimensional array of color indices, and cell array plot displays the corresponding colors in a rectangular matrix of color

```
real x=iota(-5,5)
real y=x+6
real z=outer(x,y)
plotz z x y color=green lev=12
nf
```

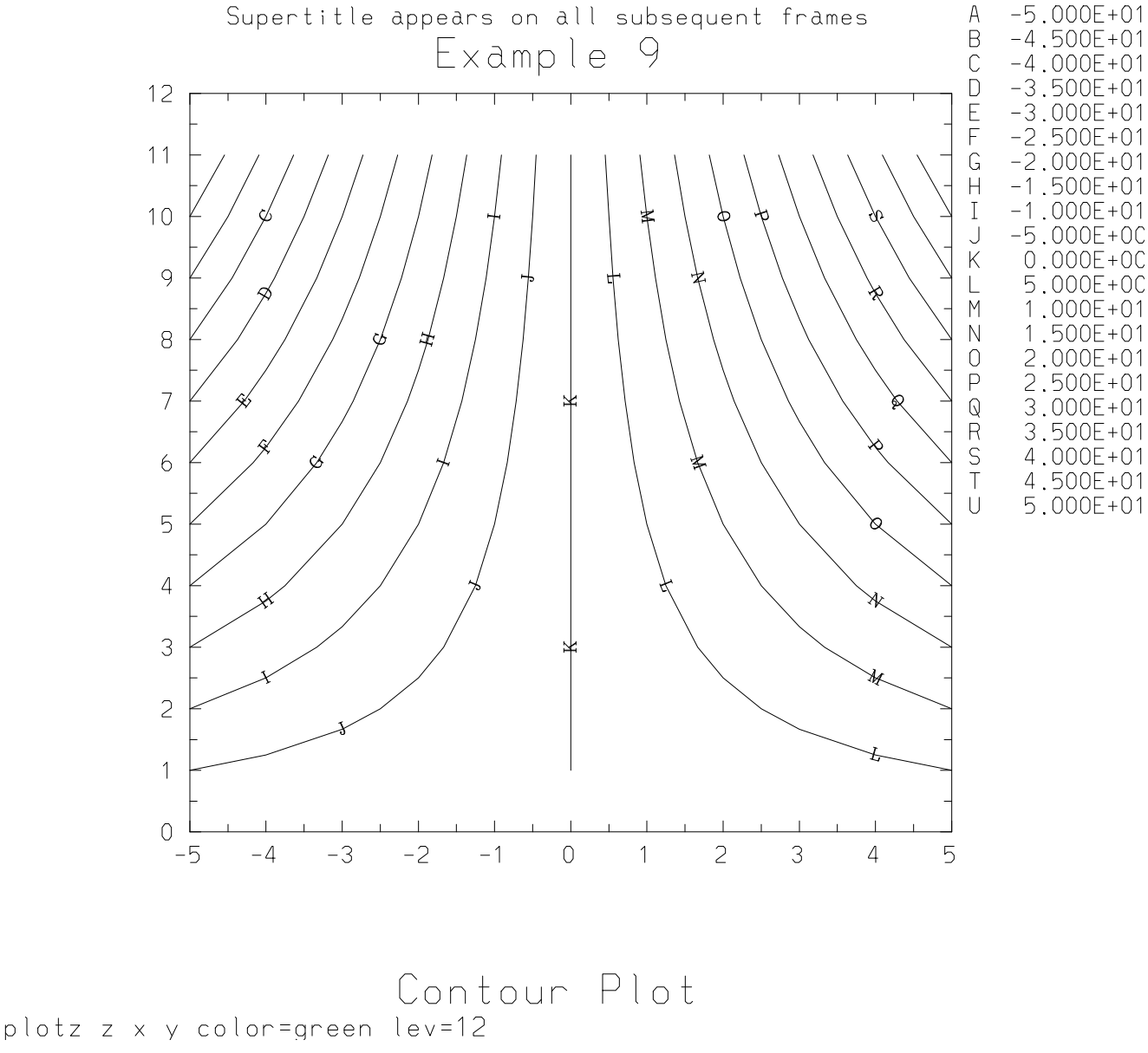


Figure 48.2: Example of contour plot with level annotation

cells. The optional arguments *xmin*,*xmax*,*ymin*,*ymax* specify actual limits for the physics data, in order to display correct *x*,*y*-labels on the plot axes. If no frame limits are given, `ploti` will use the square [0.,1.]x[0.,1.] to plot the color array.

For mesh-based data, a more realistic display may be obtained by using the `plotf` command instead; 49.3see Section 7.3 ”plotf: Fillmesh Plot” on page 57.

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified; i.e., they are not remembered across commands.

`grid`, `scale`, `font`, `legend`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

Although it is recognized, the `font` attribute currently has no effect.

48.3.1 Color-Mapping Functions

Setting the Color Map

The function `ezcscm` is used to create and install a color map with `numcols` entries. (The number of colors should be limited by the value of the EZD variable `numcol`, which defines the number of user-alterable colors.) It is called as follows:

```
ezcscm(numcols)
```

The numbers `1..numcols` then act as color indices into the map. This function should be called before plotting cell arrays to ensure that there will be colors available for the plot.

For the advanced user, the specific color map used can be altered from the default at the time that the plotting device is opened. 45.5See “Setting the Colormap” on page 20 for more information.

Mapping Real Data to Color Indices

The vector-to-color mapping function `ezcmp8` examines a real data vector, determines its maximum and minimum values, `zmin` and `zmax`, and linearly maps the colors `1..ncol` to the data. In other words, the data is partitioned into `ncol` bins and each element that falls in the same bin gets the same color. The resulting color indices are placed into the `clrndx` vector.

The function `ezcmp8ln` performs the same operation, except that the data is mapped to the colors logarithmically; i.e., the logarithm of the data is partitioned into bins with each data element colored according to the bin its logarithm falls into.

The function `ezcmp8nml` maps the data according to a normal distribution. In addition to `zmin` and `zmax`, it computes the mean `zbar` and standard deviation `zsigma` of the data. Values which

are over two standard deviations below `zbar` are mapped to color index 1; values over two standard deviations above `zbar` are mapped to color index `ncol`. Intermediate values are mapped in the normal distribution fashion.

In order to accommodate applications for which the data range may change over the course of a computation, these routines also have an input argument `zlim`, which is a two-element real array. If `zlim(1)=zlim(2)`, then `zmin` and `zmax` are computed from the data, as described above. Otherwise, `zmin` is set to the smaller of `zlim(1)` and `zlim(2)`; `zmax`, to the larger value. In this case, data values outside this range are mapped to the appropriate extreme color index.

These functions are called as follows:

- integer ncol, veclen, clrndx(veclen)
- real(Size8) data(veclen), zz(5), zlim(2)
- ezcmp8 (ncol, veclen, data, clrndx, zz, zlim)
- ezcmp8ln (ncol, veclen, data, clrndx, zz, zlim)
- ezcmp8nml (ncol, veclen, data, clrndx, zz, zlim)

The arguments have the following meaning:

`ncol` : the number of colors in the color map. Typically, this will be the value of the EZD variable `numco`
]

`veclen` : the length of the data vector. If `data` is dimensioned (`nx`,`ny`), set `veclen`= `nx*ny` and dimensi
]

`data` : the real data vector. [input: real(Size8) array]

`clrndx` : the resulting color index array. [output: integer array, same shape as `data`.]

`zz` : if zz(5)¡0 in input, the color order will be the reverse of the usual order. On output, `zz(1)=zmin`, `zz`
]

`zlim` : auxiliary input to allow user-defined limits (see discussion above). [input: real (Size8) array
]

Caution: Since `clrndx` and `zz` are output arrays, the names of these variables must be preceded by `&` if these functions are called from Basis, and the type declaration in Basis is `real(8)`, not `real(Size8)`. See the following example to see how this is done.

Older versions of EZN (before Basis 11.12) provided an `ezclr8`-family of routines, which did not have the `zlim` argument and always used the data limits. In order to provide compatibility for old applications, these are still provided as shells that call their `ezcmp8`-counterparts.

Example

The following example installs a 180-color color map, computes a function on a 100x100 mesh, maps this linearly to color index array `clrz` using the data limits, and displays the result via `ploti`.

```
integer clrNcol=180, nz=100, i, j
ezcscm(clrNcol)
real z(nz,nz), r
do i=1, nz
   do j=1, nz
      r = sqrt(1.0*i*i + 1.0*j*j) + 1e-12
      z(i,j) = sin(r) / r
   enddo
enddo
real(8) zlim(2), zz(5)
zlim(1)=0; zlim(2)=0 #To compute zmin,zmax from z.
integer clrz(nz,nz)
ezcmp8(clrNcol, nz*nz, z, &clrz, &zz, zlim)
ploti clrz
```

CHAPTER

FORTYNINE

Mesh-Oriented Commands

Mesh-oriented plots are specific for Lasnex applications. A mesh-oriented command assumes an underlying logically-rectangular two-dimensional mesh. The x-coordinate of the mesh, *xexpr*, and the y-coordinate, *yexpr*, are both two-dimensional real arrays dimensioned (`kmax,lmax`). [`kmax` and `lmax` are internal variables in the EZN package and are Basis variables in Lasnex and in Lasnex dump files.] By convention, *zone* (i,j) is the quadrilateral with upper-right corner (i,j); that is, with diagonally opposite corners (*xexpr*(i-1,j-1), *yexpr*(i-1,j-1)) and (*xexpr*(i,j), *yexpr* (i,j)).

A mesh-oriented command also requires a region map *ireg* as an argument. This is a two-dimensional integer array, also dimensioned (`kmax,lmax`), with *ireg*(i,j) the region number for zone (i,j). The values of *ireg*(1,:) and *ireg*(:,1) are irrelevant. A value of 0 indicates a "void".

The three mesh-defining arrays *xexpr*, *yexpr*, *ireg* have default names `zt`, `rt`, `ireg` respectively. If these variables are specified in the plot command, they must appear before the first key=value pair. They may be dropped from the right, with missing values replaced by defaults. Thus, "`x,color=red`" is equivalent to "`x,rt,ireg,color=red`".

A mesh-oriented command accepts attribute specifications which specify a subset of the mesh to be plotted by defining values for `krange`, `lrange`, and `region`. The command will plot the subset of the mesh consisting of zones whose indices are in the ranges specified and with region numbers in the region list.

A range specification has the form *(start:stop:inc)*. Unspecified fields in the range are set to default values, e.g. *(:stop:inc), (start::inc), (::inc), (start:), (:stop)*. `krange` specifies a range for the first subscript, and `lrange` specifies a range for the second subscript. The defaults are `krange=(1:kmax:1)` and `lrange=(1:lmax:1)`.

In the specification `region`=*region-list*, *region-list* can be a scalar or vector of integers containing a list of region numbers. The default is `region=all`, meaning all regions.

The attributes krange, lrange and region are "*sticky*", which means that after a mesh-oriented plot specifies a value for an attribute, this attribute value will stay in effect for the following mesh-oriented commands until a new frame or the attribute is reassigned another value.

For example,

```
plotm region=[1,3,5]
    #Mesh plot for regions 1, 3 and 5.
```

```
plotc te color=filled
    #The contour plot will be restricted to regions 1,3,5.
nf
```

49.1 plotm: Plotting Meshes, Boundaries, and Regions

Calling Sequence

`plotm` *xexpr,yexpr,ireg,¡keylist>*plotm ¡*keylist>*plotb ¡*keylist>*

Description

plotm is a mesh-oriented command. For general information, see the chapter introduction on 49page 47.

The `plotm` command plots meshes. If the keyword `bnd` is set to `yes` (or 1), only the boundaries of regions are plotted. If specified, *xexpr* is an array of x-axis values, *yexpr* is an array of y-axis values, *ireg* is a region map, and *¡keylist>* is a list of optional keywords and values.

If `plotm` arguments are omitted, they are supplied by using the names in the variables ezcx, ezcy, and ezcireg, respectively. Default values for these names are `zt`, `rt`, and `ireg`.

As a special case, "`plotm bnd=1`" can be abbreviated `plotb`.

By convention, the curves connecting nodes are divided into two sets,

***k-lines*:** (*xexpr*(k,:), *yexpr*(k,:)), k=1,...,`kmax`; and

***l-lines*:** (*xexpr*(:,l), *yexpr*(:,l)), l=1,...,`lmax`.

The `krange` and `lrange` attributes can be given a stride j to cause only every j'th line in that direction to be plotted. The stride is ignored for boundary plots, and ignored in drawing the lines in the opposite direction (that is, the l-lines will have all their pieces even if `krange` has a stride j, while only every j'th k-line will be plotted).

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified; i.e., they are not remembered across commands.

`grid`, `scale`, `kstyle`, `lstyle`, `thick`, `bnd`, `color`, `kcolor`, `lcolor`, `mark`, `marksize`, `labels`, `krange`, `lrange`, `region`, `legend`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See "Attribute Table" on page 31 for descriptions of the values which can be assigned to these keywords.

The default line style is `solid` and the default line thickness is 1.0. The default color is the foreground color. To override these defaults, set attributes `style`, `thick`, and `color` respectively.

The attribute mark can be used to plot markers at the nodes instead of drawing mesh lines to connect the nodes. This is similar to the command plot with the mark attribute.

Optional attributes `kstyle` and `lstyle` set the line style for the k-lines and l-lines, respectively. By default, both are set to `solid`. If a style is set to `none`, no lines are plotted in that direction.

Optional attributes `kcolor` and `lcolor` set the line color for the k-lines and l-lines, respectively. If either of these is unset, the color specified by the `color` attribute is used; if both are set, the `color` attribute is irrelevant.

Although it is recognized, the `labels` attribute currently has no effect.

Examples

The following data are used for the examples here and in Sections 49.27.2 “plotc: Plotting Contours” and 49.37.3 “plotf: Fillmesh Plot”.

```
#  Define mesh:
integer kmax=25,lmax=35 #Don't make either smaller than 25.
real xr=outer(iota(kmax),ones(lmax)),
     yr=outer(ones(kmax),iota(lmax)),
     zt=5.+xr+.2*ranf(xr),
     rt=100.+yr+.2*ranf(yr)
#  Define region map:
integer ireg(kmax,lmax)=1
     ireg(1,)=0
     ireg(,1)=0
     ireg(2:15,8:12)=2
     ireg(2:15,13:lmax)=3
     ireg(4:7,4:7)=0 #Define an internal void.
integer k2=3,l2=10   #Index of a point in region 2.
#  Define data on the mesh:
real s=1000., z=s*(rt+zt)
     z(4:12,4:10)= z(4:12,4:10)*.9
     z(6,6)=z(6,6)*.9
     z(16:18,18:22)=z(16:18,18:22)*1.2
     z(17,17)=z(17,17)*1.1
```

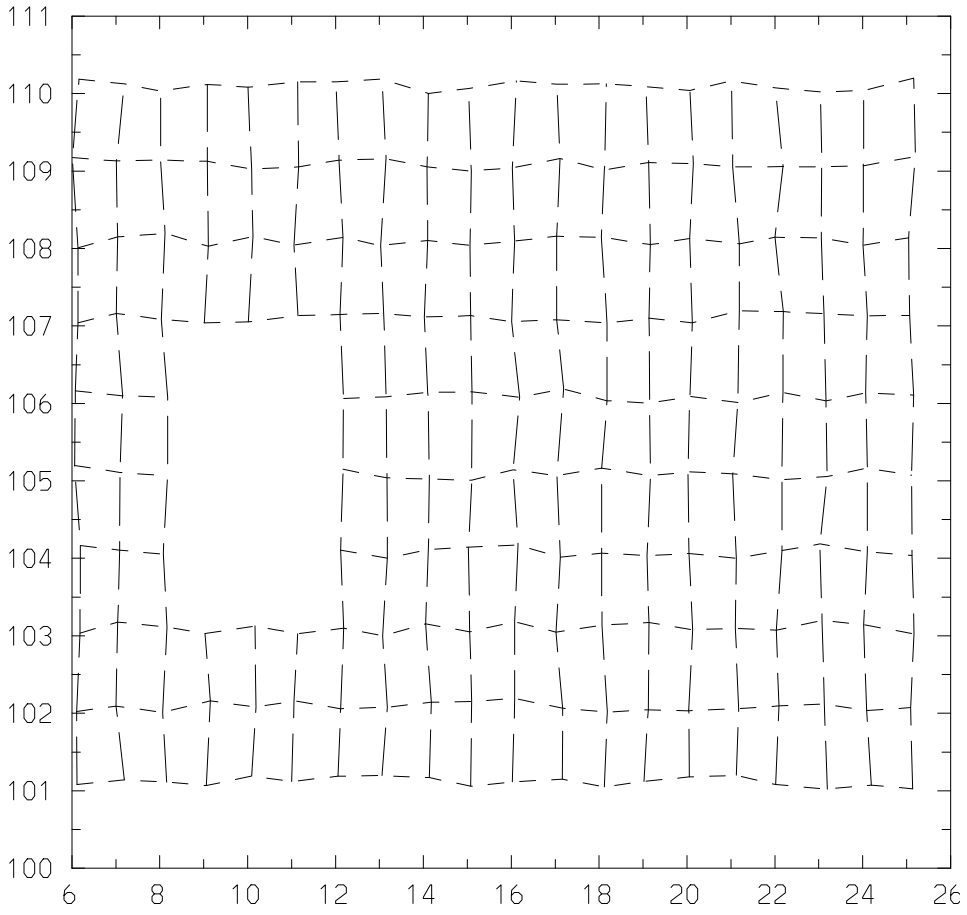
In the first example, a mesh is plotted with k-lines dashed and l-lines dotted. Here, the displayed mesh has been restricted to lines with k ranging from 1 to 20 and l from 1 to 10. Note that nothing is plotted where the interior void was defined.

Here we plot just two regions. Note that the full extent of the mesh is used.

And here we plot all region boundaries, and then just the l-lines:

Finally, we plot all region boundaries, and mark region 2 with text in it. Note that this looks better on the screen, because the colored mesh lines make the text stand out.

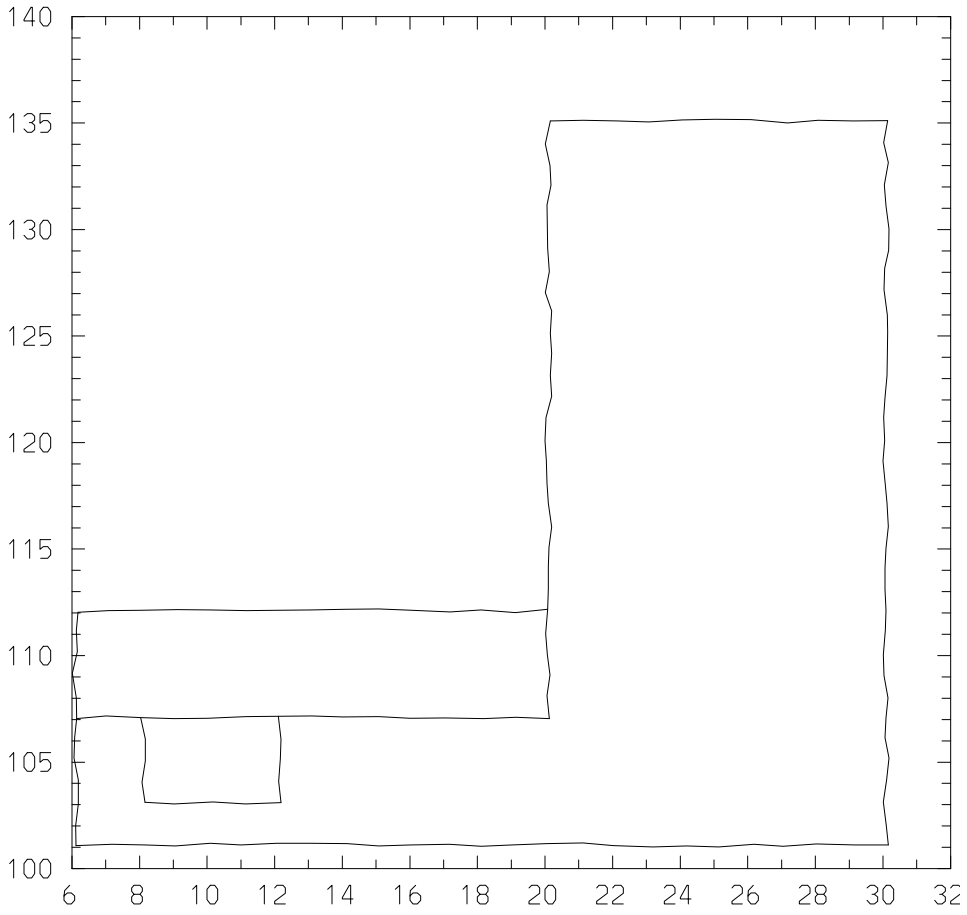
```
plotm kstyle=dashed,lstyle=dotted,krange=1:20,lrange=1:10
nf
```




```
plotm kstyle=dashed lstyle=dotted krange=1:20 lrange=1:10
```

Figure 49.1: Example of mesh plot

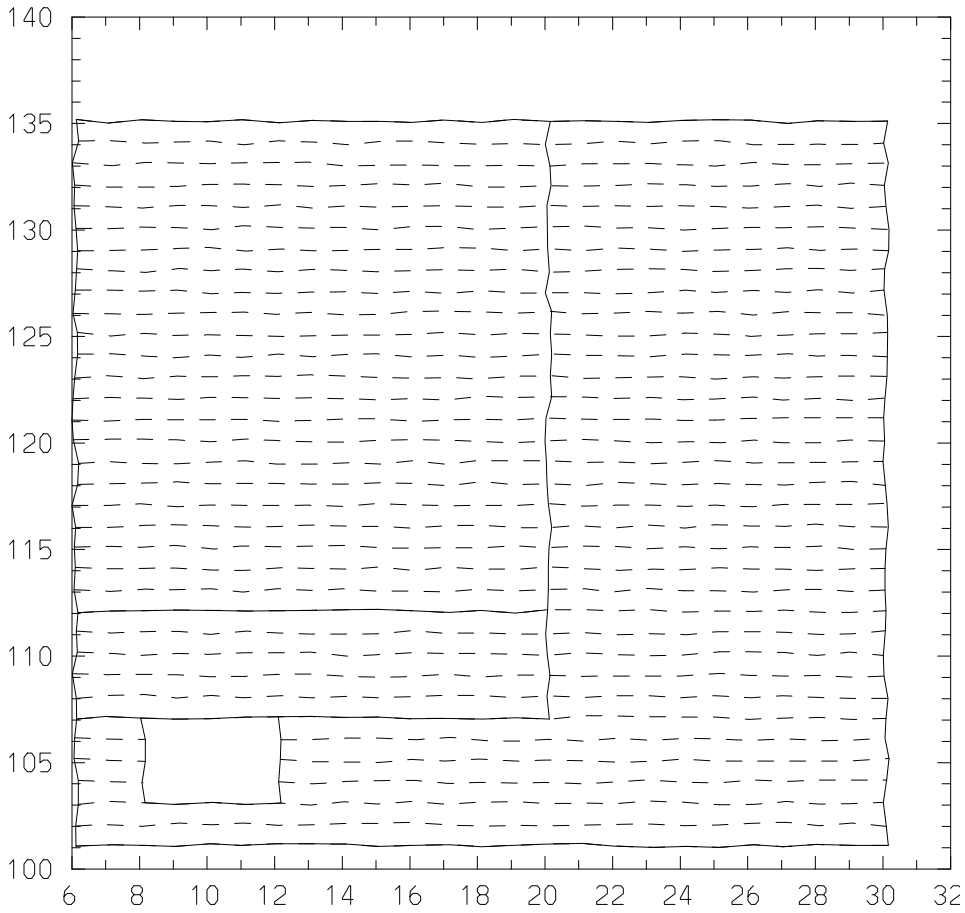
```
plotm bnd=1,region=[1,2] # Plot boundaries of regions 1 and 2.
nf
```




```
plotm bnd=1 region=[1,2]
```

Figure 49.2: Example of boundaries plot

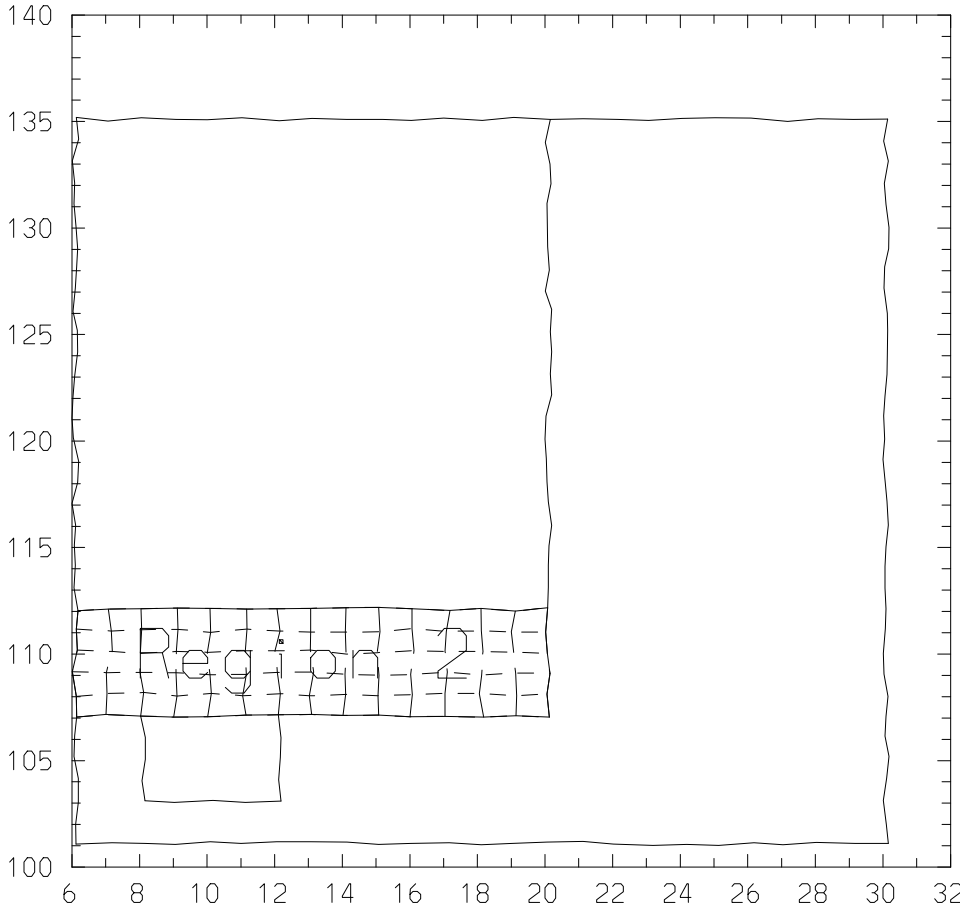
```
plotb               # Plot boundaries.
plotm kstyle=none lstyle=dotted
                    # Plot just the l-lines of the mesh.
nf
```




```
plotm bnd=1
plotm kstyle=none lstyle=dotted
```

Figure 49.3: Example of l-lines plot

```
plotb               # Plot boundaries
plotm region=2 kstyle=dashed lstyle=dotted color=green
text "Region 2" zt(k2,l2) rt(k2,l2) 32
nf
```




```
plotm bnd=1
plotm region=2 kstyle=dashed lstyle=dotted color=green
text "Region 2" zt(k2,l2) rt(k2,l2) 32
```

Figure 49.4: Example of plotting a region with text

49.2 plotc: Plotting Contours

Calling Sequence

```
plotc <fexpr>,<xexpr>,<yexpr>,<ireg>,<keylist>
plotc <fexpr>,<keylist>
```

Description

plotc is a mesh-oriented command. For general information, see the chapter introduction on 49page 47.

The `plotc` command plots a contour map of *fexpr* above the mesh described by *xexpr* and *yexpr*. *fexpr* is a two-dimensional array of values dimensioned the same as *xexpr* and *yexpr*. If specified, *xexpr* is an array of x-axis values, *yexpr* is an array of y-axis values, *ireg* is a region map, and *¡keylist>* is a list of optional keywords and values. Strides in krange or lrange are ignored by `plotc`.

If `plotc` arguments are omitted, they are supplied by using the names in the variables ezcx, ezcy, and ezcireg, respectively. Default values for these names are `zt`, `rt`, and `ireg`.

fexpr can also be the name of a function or macro which, when called with no arguments, returns a two-dimensional array of values of the appropriate shape.

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified, i.e. they are not remembered across commands.

`grid`, `scale`, `thick`, `style`, `font`, `mark`, `marksize`, `lev`, `color`, `krange`, `lrange`, `region`, `legend`, `point`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

Although it is recognized, the `font` attribute currently has no effect.

Contour Levels,Colors, etc.

The discussion of the command `plotz` (48.2see Section 6.2 ”plotz: Plotting Contours” on page 39) contains a detailed explanation of the way contour levels and colors are specified. The discussion there applies to `plotc` as well.

The primary difference between `plotc` and `plotz` is that the former is a mesh-oriented command. This means that only the `plotz` mesh data discussion applies to `plotc`. Furthermore, because of the underlying mesh and the associated region map, the `plotc` command has the possibility of controlling the subregion over which contours are displayed by use of attributes `krange`, `lrange`, `region`.

The `plotc` command assumes that the physics quantity *fexpr* is zone-based, or zone-centered, which means that *fexpr*(i,j) is the average value associated with zone (i,j), and the values of *fexpr*(1,:) and *fexpr*(:,1) are irrelevant. Since the contour plot requires data at the mesh points, these values are interpolated to the mesh by simple averaging (as permitted by the subregion specification).

To plot contours of an actual mesh-based, or point-centered, quantity (data at the mesh points), such as `ut` or `vt`, use the specification "`point=yes`" to tell `plotc` that the values are mesh-based. This will avoid the above-mentioned averaging. (The default is `point=no`.) In the case of a mesh-based variable on a mesh containing no interior voids, `plotz` and `plotc` are equivalent. Thus,

```
plotc z,x,y,ireg,point=yes,<keylist>
```

is equivalent to

```
plotz z,x,y,<keylist>
```

where ¡*keylist*> contains attributes allowed by both commands.

Customizing Contour Plots

Several control variables which may be used to customize the contour plots were discussed in 48.2.2"Contour Control Parameters" on page 40 of the `plotz` description. Some NCAR Conpack parameters can be set to futher customize the contour plots. The routines cpseti, cpsetr, and cpsetc are used to set these parameters. The inquiry routines cpgeti, cpgetr and cpgetc are used to find out the current setting of those parameters. The user should refer to the NCAR document "CONPACK, A Contouring Package" (available on the web at http://ngwww.ucar.edu/ngdoc/ng/supplements/conpack/) for detailed information.

The NCAR Conpack displays dotted lines around "voids". (See **??**Figure 7.5 on page 56 for an example.) These can be eliminated by executing the following calls prior to the `plotc` command:

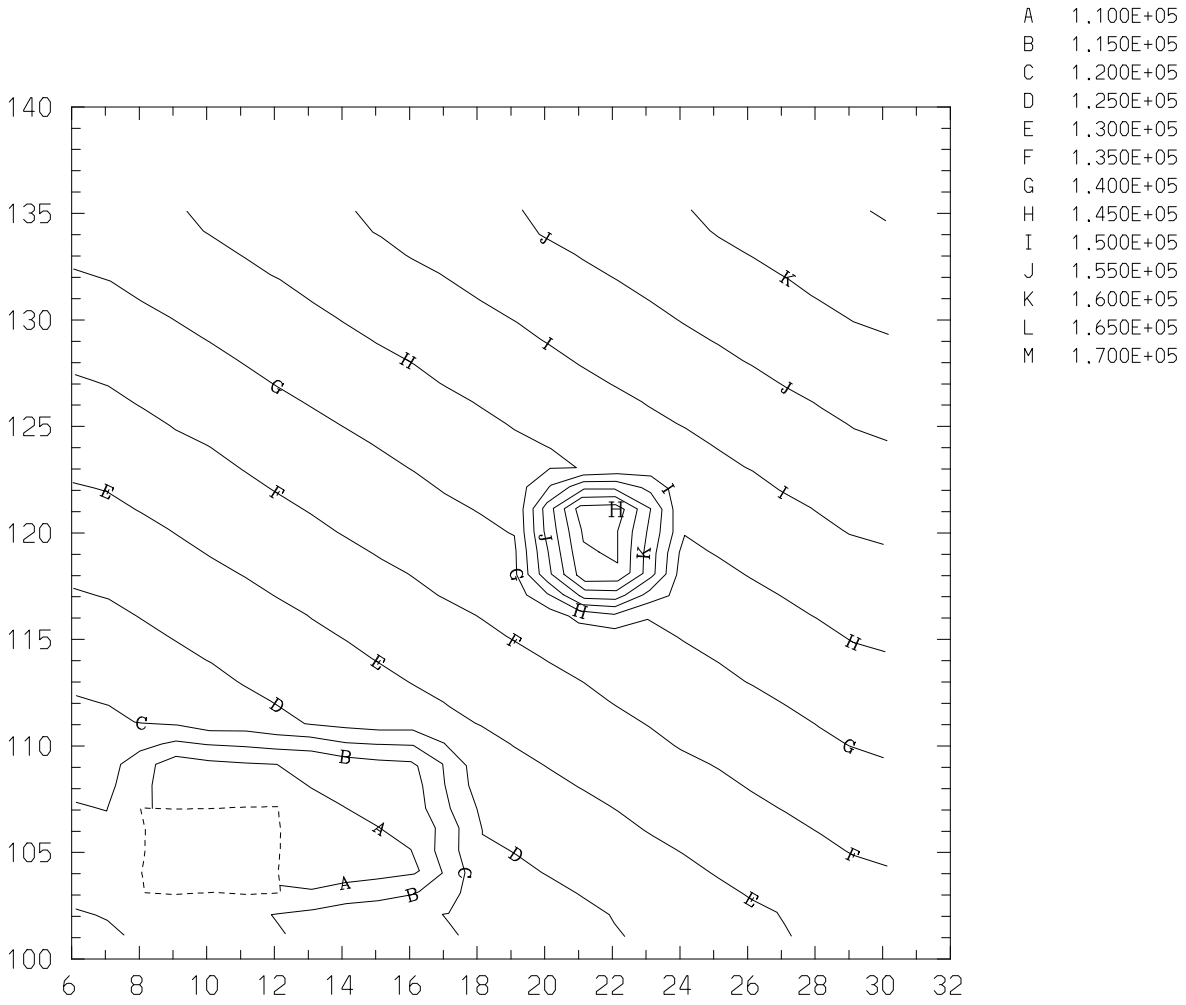
```
call cpseti ("PAI", -2)
call cpseti ("CLU", 0)
call cpseti ("PAI", -3)
call cpseti ("CLU", 0)
```

The first pair of calls turns off the boundary of an area filled with the "special value" used to indicate missing data. The second pair treats the case of the internal mapping routine returning "out of range", say from the `rt=-1.e8` conventionally used in voids.

Example

The following is an example of using `plotc` with default arguments. The data are as defined before the `plotm` examples, 49.1page 49. Note the dotted lines around the internal void.

```
plotc z
```




```
plotc z
```

Figure 49.5: Example of mesh contour plot

49.3 plotf: Fillmesh Plot

Calling Sequence

`plotf` *pvar*,*xexpr*,*yexpr*,*ireg*,¡*keylist*> `plotf` *pvar*,¡*keylist*> `plotf` *cindex*,¡*keylist*>

Description

`plotf` is a mesh-oriented command. For general information, see the chapter introduction on 49page 47.

The

`plotf` command plots a color-filled mesh which displays the physics quantity *pvar* in the zones of interest with colors. If specified, *xexpr* is an array of x-axis values, *yexpr* is an array of y-axis values, *ireg* is a region map, and ¡*keylist*> is a list of optional keywords and values.

If `plotf` arguments are omitted, they are supplied by using the names in the variables ezcx, ezcy, and ezcireg, respectively. Default values for these names are `zt`, `rt`, and `ireg`.

pvar can also be the name of a function or macro which, when called with no arguments, returns a two-dimensional array of values of the appropriate shape.

The colors assigned to the individual zones range from the beginning color in the colormap (after the "*named colors*" *red, green, blue, yellow*, etc.) to the last color in the colormap. The color varies from low color index to high color index as *pvar* varies from its minimum to maximum values. (This order can be reversed, as described below.)

The mapping of colors can be *linear*, *logarithmic*, or *normally distributed*. The user can use the attribute cscale to specify the mapping choice. For example, set `cscale=log` to set the color mapping to logarithmic values of the physics quantity. The default mapping (or `cscale= lin`) is linear. The normal distribution color mapping (`cscale=normal`) will map *pvar* values which are over two standard deviations below the mean to the lowest color index, and *pvar* values which are over two standard deviations above the mean to the highest color index. Intermediate *pvar* values are mapped in the normal distribution fashion. A colored annotation on the right side of the frame displays the assignment of colors to the corresponding values of *pvar*.

To reverse the order of the color mapping, precede one of the above `cscale` options with the letter "`r`": `rlin`, `rlog`, `rnormal`.

The attribute `zlim=[`*zmin*`,`*zmax*`]` allows the user to specify limits to be used when mapping physical values to colors (by any of the above-mentioned color scales). If not supplied, the minimum and maximum values in *pvar* are used. The use of `zlim` allows one to use the same colormap for a series of related plots, such as the time-evolution of *pvar*.

The `plotf` command also accepts an integer array *cindex* to directly assign color indices to the zones in the mesh. The integer array must be of dimension (`kmax`,`lmax`) and contain values between the lowest color index and the highest color index (usually the range 1 to 192). When directly assigned color indices are used, no color annotation will be displayed, because the EZN package has no knowledge of how the color mapping was defined.

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified, i.e. they are not remembered across commands.

`cscale`, `krange`, `lrange`, `region`, `legend`, `point`, `zlim`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

Although it is recognized, the `point` attribute currently has no effect.

Due to the possibility of different color assignment schemes in different regions or with different physics quantities, the *krange, lrange, region* attributes are made *”non-sticky*”; i.e., the submesh specifications will not be remembered during subsequent fillmesh plots in the same frame. This differs from the effects of `krange`, `lrange`, `region` on the `plotm` command (49.1see Section 7.1 ”plotm: Plotting Meshes, Boundaries, and Regions” on page 48).

49.3.1 Fillmesh Level Annotation

When `plotf` is invoked with an array of physics quantities, a display is given to the right of the plot to associate the colors with physical values. Setting ezcfmkey=off will cause the frame not to display the fillmesh level annotation. (However the portion of the frame for level annotation is still allocated. To utilize the whole frame without level annotation, the variables ezccntfr and ezcfixed need be set properly). The default is ezcfmkey=on.

The variable ezcfmfill specifies either *solid* color fill or *hollow* fill (i.e. just color the border) for each cell containing the numerical annotation. ezcfmfill=“solid” (the default) specifies the solid fill; any other value, e.g. ezcfmfill=“hollow”, will make a hollow fill.

49.3.2 Color-Mapping Functions

The user who wants to customize the color mapping in a fillmesh plot may wish to 48.3.1see Section 6.3.1 ”Color-Mapping Functions” on page 44 for information on the `ezcmp8`-family of functions which produce a `cindx` array directly from the data. Older versions of EZN (before Basis 11.12) provided an `ezcfmc`-family of routines, which did not have the `zz` output array. For compatibility with old applications, these are still provided as shells that call their `ezcmp8`-counterparts.

As an alternative, the user may wish to directly call the routines used by the `plotf` command, namely `ezclrm` for a linear mapping, `ezclrmln` for a logarithmic mapping, or `ezclrmnml` for a normal distribution mapping. These are similar to their `ezcmp8`-counterparts, except that the region map `ireg` is passed as an argument and zones for which `ireg=0` are ignored.

They are called as follows:

- integer ncol,kmax,lmax,ireg(kmax,lmax),cindx(kmax,lmax)

- real(Size8) z(kmax,lmax), zz(5), zlim(2)
- ezclrm (ncol, z, ireg, kmax, lmax, cindx, zz, zlim)
- ezclrmln (ncol, z, ireg, kmax, lmax, cindx, zz, zlim)
- ezclrmnml (ncol, z, ireg, kmax, lmax, cindx, zz, zlim)

The arguments are defined as follows:

ncol : the number of colors requested. (Usually use EZD variable `numcol`.) [input: integer]

z : an array of physics quantity values. [input: real(Size8) array]

ireg : the associated region map (see introductory section, 49page 47). [input: integer array, same shape as `z`]

kmax : the first dimension of the `z`, `ireg`, and `cindx` arrays. [input: integer]

lmax : the second dimension of the `z`, `ireg`, and `cindx` arrays. [input: integer]

cindx : resulting array of color indices. [output: integer array, same shape as `z`]

zz : if zz(5)¡0 in input, the color order will be the reverse of the usual order. On output, `zz(1)=zmin`, `zz(2)=zmax`, `zz(3)=zbar` (set only by `ezclrmnml`), `zz(4)=zsigma` (set only by `ezclrmnml`), `zz(5)`is the mapping type: 0 for linear, 1 for logarithmic, 2 for normal. This is used in generating the fillmesh level annotation. [inout: real (Size8) array]

zlim : the (optional) limitations for the z values; `zlim(1)=zmin`, `zlim(2)=zmax`. (As with `ezcmp8`, if these are equal, use the data limits.) [input: real(Size8) array]

Caution: Since `cindx` and `zz` are output arrays, the names of these variables must be preceded by & if these functions are called from Basis. Note also that the type `real(Size8)` is only recognized in a MPPL Fortran source code. Use `real(8)` when calling these functions from Basis. (See the final example, below.)

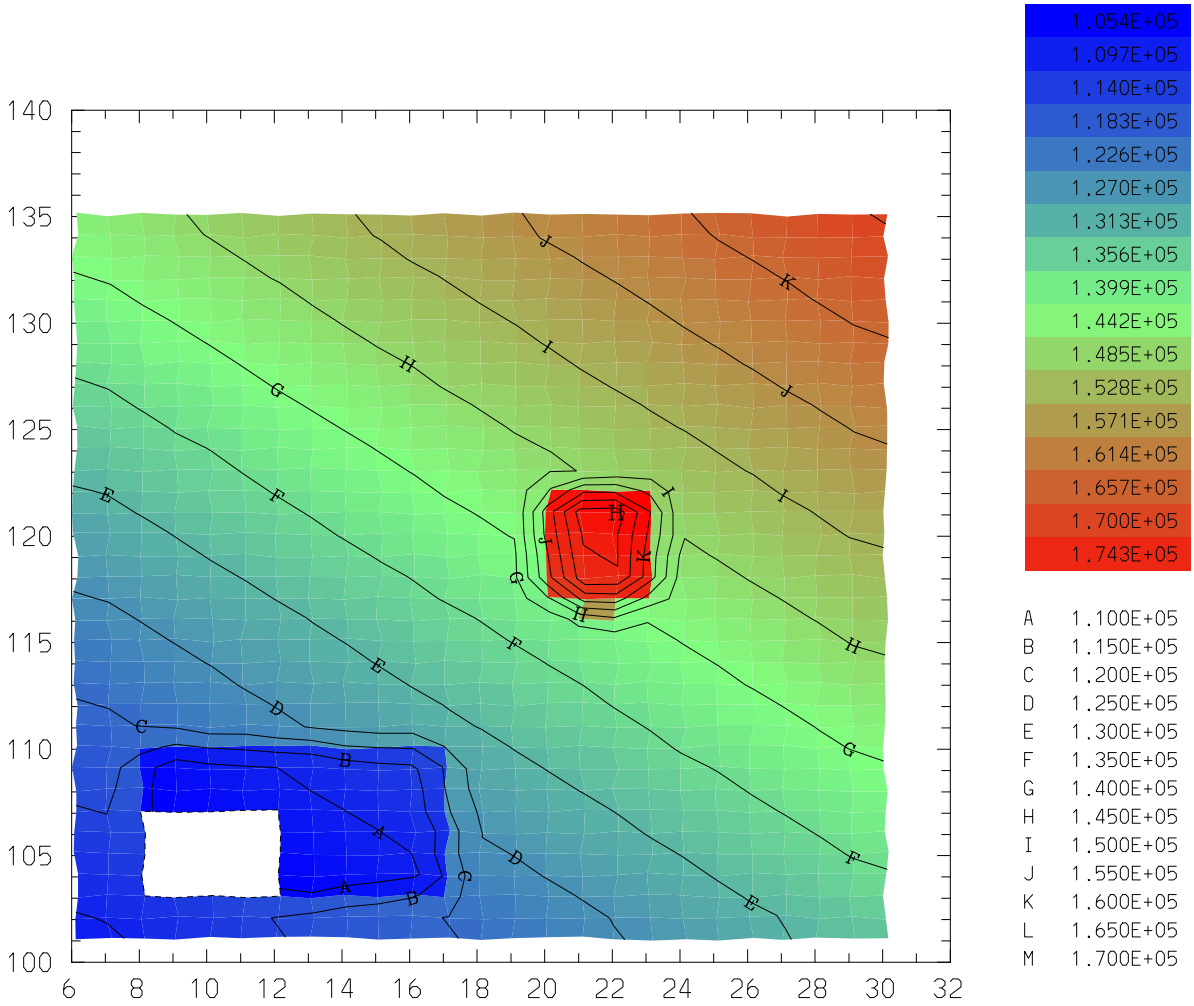
For the advanced user, the specific color map used can be altered from the default at the time that the plotting device is opened. 45.5See “Setting the Colormap” on page 20 for more information.

Examples

For our first example, assume the same data as defined before the `plotm` examples, 49.1page 49. Note that `plotf` plots nothing in the void, so it has the background color.

The result of these commands is shown in **??**Figure 7.6 on page 60. Note the shift in location of the contour annotation from **??**Figure 7.5 on page 56.

```
plotf z
plotc z  # Superimpose contours
nf
```




```
plotf z
plotc z
```

Figure 49.6: Example of fillmesh plot

For our next set of examples, assume that a Lasnex dump file `test1z` has been created, and we want to examine some of its physics variables in Sod. First we do a linearly-scaled fillmesh plot of variable `te`:

```
open test1z
plotf te
nf
```

Next we do a logarithmically-scaled fillmesh plot of variable `ti`:

```
plotf ti cscale=log
nf
```

Finally, we use the color-mapping function `ezclrm` to generate a color index array and plot that. Because we have used the data limits in `zext`, this is essentially the same as the previous plot of `te`, except that it has no color-mapping legend.

```
integer nndx(kmax,lmax)
real(8) zz(5),zext(2)
zext(1)=min(te)
zext(2)=max(te)
ezclrm(numcol,te,ireg,kmax,lmax,&nndx,&zz,zext)
plotf nndx
```

49.4 plotv: Plotting Vectors

Calling Sequence

`plotv` *xexpr,yexpr,xvexpr,yvexpr,ireg,¡keylist>* `plotv` *¡keylist>*

Description

`plotv` is a mesh-oriented command. For general information, see the chapter introduction on 49page 47.

The

`plotv` command plots velocity vectors on a mesh. If specified, *xexpr* is an array of x-axis values, *yexpr* is an array of y-axis values, *xvexpr* is the displacement for *xexpr*, *yvexpr* is the displacement for *yexpr*, *ireg* is a region map, and *¡keylist>* is a list of optional keywords and values.

If `plotv` arguments are omitted, they are supplied by using the names in the variables `e`zcx, ezcy, ezcxv, ezcyv, and ezcireg, respectively. Default values for these names are `zt`, `rt`, `vt`, `ut`, and `ireg`. (*Caution*: Note that `vt` is the velocity in the x-direction; `ut`, the y-direction.)

A series of arrows from (*xexpr, yexpr*) to (*xexpr+xvexpr*dx*, *yexpr+yvexpr*dy*) is plotted. The values *dx* and *dy* are chosen so that the maximum extent of an arrow in the corresponding direction

is the frame size in that direction multiplied by the `vsc` attribute. (See also variable `ezcvsc`.) The default for `vsc` is .05; this default can be changed by assigning a new value to defvsc.

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified, i.e. they are not remembered across commands.

`grid`, `scale`, `thick`, `vsc`, `color`, `krange`, `lrange`, `region`,

`legend`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

The default line thickness is 1.0 and the default color is the foreground color. To override these defaults, set attributes `thick` or `color`, respectively.

Examples

In the first example, the input arrays are explicitly specified. The line thickness of vectors will be 2.0.

```
real vx(10,8),vy(10,8),x(10,8),y(10,8)
```

In the second example, the default names `zt`, `rt`, `vt`, `ut`, and `ireg` are used. The displacement vectors are scaled to 0.08 of the frame size. (Note that the vectors are longer and thinner.) Only vectors originating at nodes of zones in regions 1 and 4 are plotted.

Customizing Vector Plots

We have already discussed variables `ezcvsc` and `defvsc`, which may be used to control vector plots. Some NCAR Vectors package parameters can be set to futher customize the vector plots. The routines vvseti, vvsetr, and vvsetc are used to set these parameters. The inquiry routines vvgeti, vvgetr and vvgetc are used to find out the current setting of those parameters. The user should refer to the NCAR document “Vectors, A Vector Field Plotting Utility” (available on the web at http://ngwww.ucar.edu/ngdoc/ng/supplements/vectors/) for detailed information.

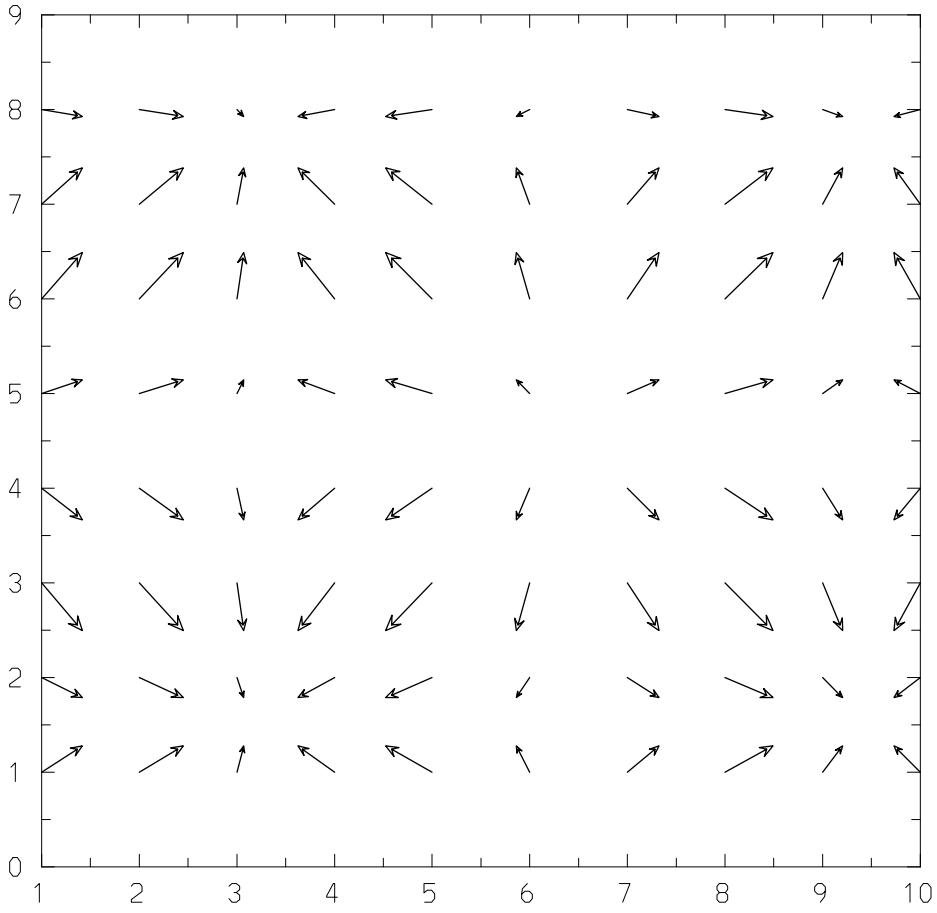
The NCAR Vectors package displays the magnitude of the largest vector plotted in the lower right-hand corner of the frame, as in the examples. To obtain the value displayed for the maximum vector length, do the following (and don’t forget the ampersand):

```
real vecmaxcall vvgetr ("VMX", \&vecmax)
```

To move the maximum vector display closer to the picture, execute the following calls prior to the `plotv` command:

```
call vvseti ("CPM", -2)
call vvsetc ("MNT", " ")
```

```
integer i,j, ireg(10,8)
do i=1,10;do j=1,8
   x(i,j)=i; y(i,j)=j
   vx(i,j)=sin(i); vy(i,j)=cos(j)
   enddo;enddo
# Define regions:
ireg(2:5,2:4)=1
ireg(2:5,5:8)=2
ireg(6:10,2:4)=3
ireg(6:10,5:8)=4
plotv x,y,vx,vy,ireg,thick=2.0 # Arguments explicitly specified.
```

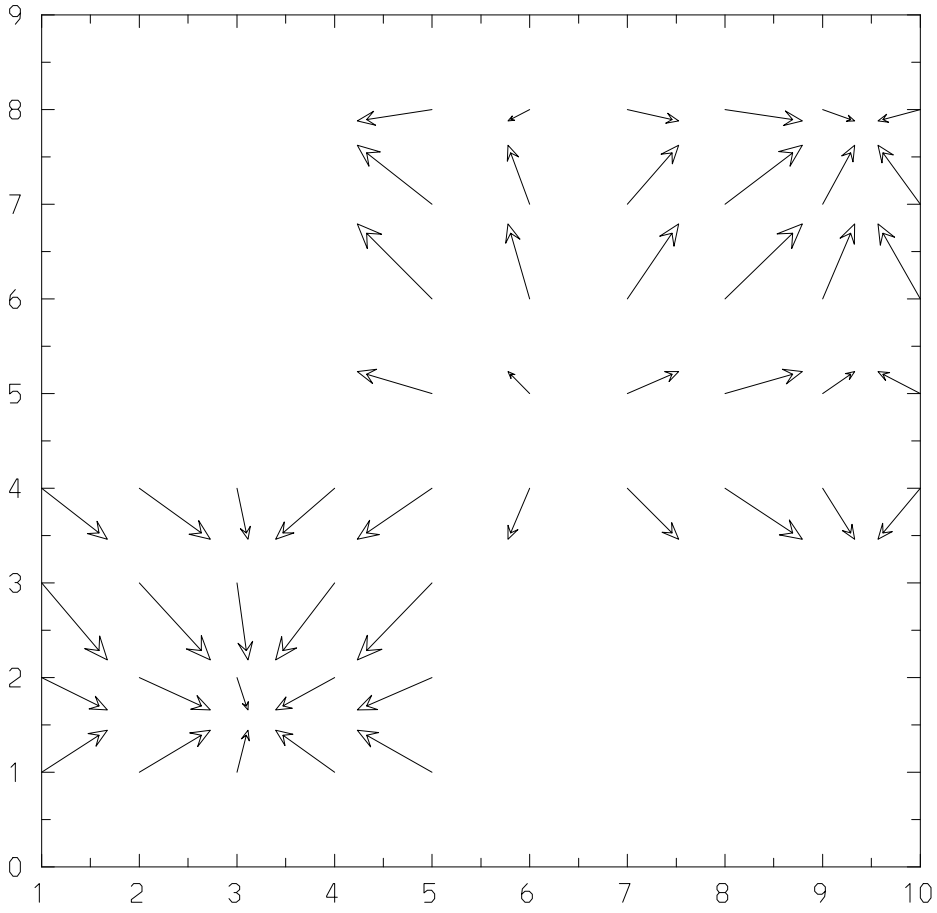



```
plotv x y vx vy ireg thick=2.0
```

0.140E+01
MAXIMUM VECTOR

Figure 49.7: Example of plotv

```
# Continuation from the last example.
nf
# Set up zt,rt,vt,ut:
real zt=x
real rt=y
real vt=vx
real ut=vy
plotv vsc=.08 region=[1,4]
```



plotv vsc=.08 region=[1,4]

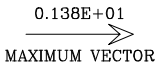


Figure 49.8: Another plotv example

```
call vvsetr ("MXX", 0.9)
call vvsetr ("MXY", -.15)
```

The first call is required to change from the default “compatibility mode” so that the values set by the following calls take effect. Without the second call, the magnitude of the minimum vector is displayed near the bottom center of the plot area; this call turns it off. The next two calls set the *x*- and *y*-coordinates of the maximum vector display, relative to the plot area. The default values are approximately 1.02 and -.35, respectively.

```
call vvsetc("MXT", "MY OWN LABEL")
```

will change the “MAXIMUM VECTOR” text to “MY OWN LABEL”.

49.5 plotr: Lasnex Rayplots

Calling Sequence

`plotr` *lasernum,¡keylist>*

Description

`plotr` is a command which only works in Lasnex or when examining a Lasnex dump file with Sod. The `plotr` command plots rays of the laser number specified. The ray is numbered and if the disposition marks were defined, the special marks will be plotted at the ends of the rays. The ray is plotted with arrows along the path to indicate the direction it travels. *lasernum* is the laser number to be plotted and *¡keylist>* is a list of optional keywords and values.

Three control variables ezcraylab, ezcarsp, ezcarsz are provided to customize the appearance of the ray plots. The variable `ezcraylab` can be set to “`on`” or “`off`” to control whether ray labels are plotted or not. The variable `ezcarsp` is a multiplier to the default arrow spacing in the plot. The user specifies this multiplier to extend or to shrink the spacing between arrows. The variable `ezcarsz` is the multiplier to the default arrow size. For example, if `ezcarsz` is set to 0.75, then the arrows plotted will be 75% of the default size.

`plotr` by itself plots the entire ray path, which starts a very long way from the physical mesh. It can be used in conjunction with a `frame` command or with a mesh-oriented command, which will delineate the region of interest.

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified, i.e. they are not remembered across commands.

`grid`, `scale`, `thick`, `color`, `legend`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

The default line thickness is 1.0. The default color is the foreground color. To override these defaults, set attributes `thick` and `color` respectively.

Ray power

The power of each ray at each point is stored in an array named `rayppow`. This value can be used to color each ray’s trajectory depending upon the ray’s power at each point. This is done by setting the attribute `color=power`.It is often more useful to color the ray according to the relative power remaining “on” the ray; namely, (raypow(i)-min(raypow))/ (max(raypow)-min(raypow)), where the min and max are restricted to the ray being plotted. If `color=relpow`, the color indicates the relative power for each ray. Setting `ezcthickray=on` causes `plotr` to use ray thickness to indicate initial (maximum) power relative to the average maximum power of all rays. However, line thickness is not a very sensitive diagnostic.

When ray power coloring is in effect, a display is given to the right of the plot to associate the colors with physical values. Setting ezcpwkey=off will cause the frame not to display the power level annotation. (However the portion of the frame for level annotation is still allocated. To utilize the whole frame without level annotation, the variables ezccntfr and ezcfixed need be set properly). The default is ezcfpwkey=on.

The variable ezcpwfill specifies either *solid* color fill or *hollow* fill (i.e. just color the border) for each cell containing the numerical annotation. ezcpwfill=“solid” (the default) specifies the solid fill; any other value, e.g. ezcpwfill=“hollow”, will make a hollow fill.

Examples

Assume a Lasnex dump file `test2z` containing laser data has been created, and we want to do post analysis about lasers in Sod:

```
open test2z
plotm
plotr 1
nf

ezcarsp=2.  # Set the arrow spacing twice as far as the default.
ezcarsz=0.8 # Set the arrow size 80% of the default size.
plotm
plotr color=rainbow     # If no lasernum is given, default to 1.
        # color=rainbow makes the rays different colors
        # for easy identification.
```

CHAPTER

FIFTY

Polygonal-Mesh Commands

In addition to the logically-rectangular (k,l)-meshes discussed in the previous chapter, EZN also provides some support for arbitrary polygonal meshes (starting with Basis 11.12). A polygonal mesh is simply a collection of polygons defined by two arrays containing the x- and y-coordinates of the polygon vertices. If these are one-dimensional arrays, then a single polygon is defined. If they are two-dimesional arrays, dimensioned (*npts,npoly*), then *npoly npts*-sided polygons are being provided. Note that polygons with fewer than *npts* vertices can be included in the collection by repeating the last point as many times as necessary to fill in the *npts* coordinates.

50.1 plotp: Plotting Polygonal Meshes

Calling Sequence

```
plotp x,y,<keylist>
```

Description

The `plotp` command plots a polygonal mesh by filling the polygons with a specified color. If specified, ¡*keylist*> is a list of optional keywords and values.

Note that no assumptions are made about the connectivity of the collection. If there are overlapping polygons, those appearing later in the list will overplot those plotted earlier.

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified; i.e., they are not remembered across commands.

`grid`, `style`, `color`, `legend`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

The default color is the foreground color. To override this default, set attribute `color`=*mycolor*. This command has no concept of regions, but you can achieve this same effect by issuing several `plotp` commands on the same frame, using different colors for different regions.

The optional argument `style=`mystyle can be used to modify the appearance of the plot. The default value for mystyle is solid. If any other value, such as hollow, is given for mystyle, only the boundaries of the polygons will be plotted. In this case, `plotp` is analogous to the use of `plotm` for a (k,l)-mesh.

Examples

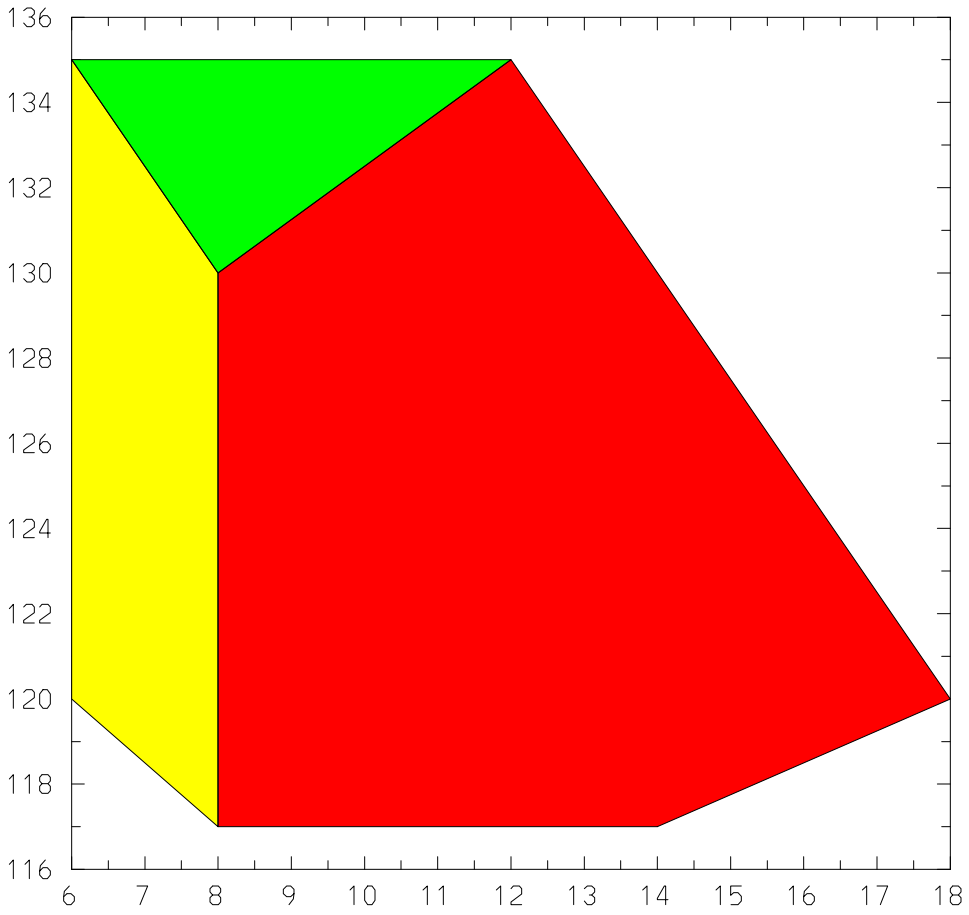
The following code defines a polygonal mesh consisting of a pentagon, a quadrilateral, and a triangle which fill up an irregular hexagon.

```
nf; ezcshow=false
real x5(5,3), y5(5,3)
x5(:,1) = [ 14.,  8.,  8., 12., 18.]
y5(:,1) = [117.,117.,130.,135.,120.]
x5(:,2) = [  8.,  6.,  6.,  8.,  8.]   # Repeat fourth point.
y5(:,2) = [117.,120.,135.,130.,130.]   # Repeat fourth point.
x5(:,3) = [  6.,  8., 12., 12., 12.]   # Repeat third point twice.
y5(:,3) = [135.,130.,135.,135.,135.]   # Repeat third point twice.
# First plot the parts individually, in different colors:
plotp x5(1:5,1) y5(1:5,1) color=red
plotp x5(1:4,2) y5(1:4,2) color=yellow
plotp x5(1:3,3) y5(1:3,3) color=green
sf
# Superimpose full mesh, boundaries only, in the foreground color.
plotp x5 y5 style=hollow
sf
```

The resulting picture appears in **??**Figure 8.1 on page 71.

The second example again assumes the same data as for the `plotm` examples, 49.1page 49. We illustrate the use of `plotp` to solid-fill the regions of a mesh with different colors.

```
nf; ezcshow=false
# Read standard Basis utility file to get gather1 function.
read Utilities
# Cycle through the regions, plotting each a different color,
# starting with color 2 in the standard color list.
integer i, kalm, nreg
character*16 mycolor
character*40 mylegend
kalm = kmax*lmax
integer iregx(kalm) = shape( where(ireg==0,-1,ireg), kalm)
# The above set posititons corresponding to ireg=0 to -1.
```




```
plotp x5(1:5,1) y5(1:5,1) color=red
plotp x5(1:4,2) y5(1:4,2) color=yellow
plotp x5(1:3,3) y5(1:3,3) color=green
plotp x5 y5 style=hollow
```

Figure 50.1: Example of polygonal-mesh plot

```
nreg = max(ireg)
do i=1,nreg
   integer izon = where( iregx=i, iota(kalm) )
   integer nz = length(izon)
   if (nz==0) next  # Omit empty regions.
   mycolor = color(i+1)  # This will work only if < 17 regions.
   mylegend = "Region "//format(i,0)//" is colored "//mycolor
   integer iq(4,nz)
   iq(1,) = izon
   iq(2,) = izon-1
   iq(3,) = izon-kmax-1
   iq(4,) = izon-kmax
   real xq(4,nz) = gather1( shape(zt,kalm), iq)
   real yq(4,nz) = gather1( shape(rt,kalm), iq)
   # Color the i-th region.
   plotp xq yq color=mycolor legend=mylegend
enddo
sf
```

50.2 plotpf: Polygonal Fillmesh Plot

Calling Sequence

```
plotpf pvar,x,y,<keylist>
```

Description

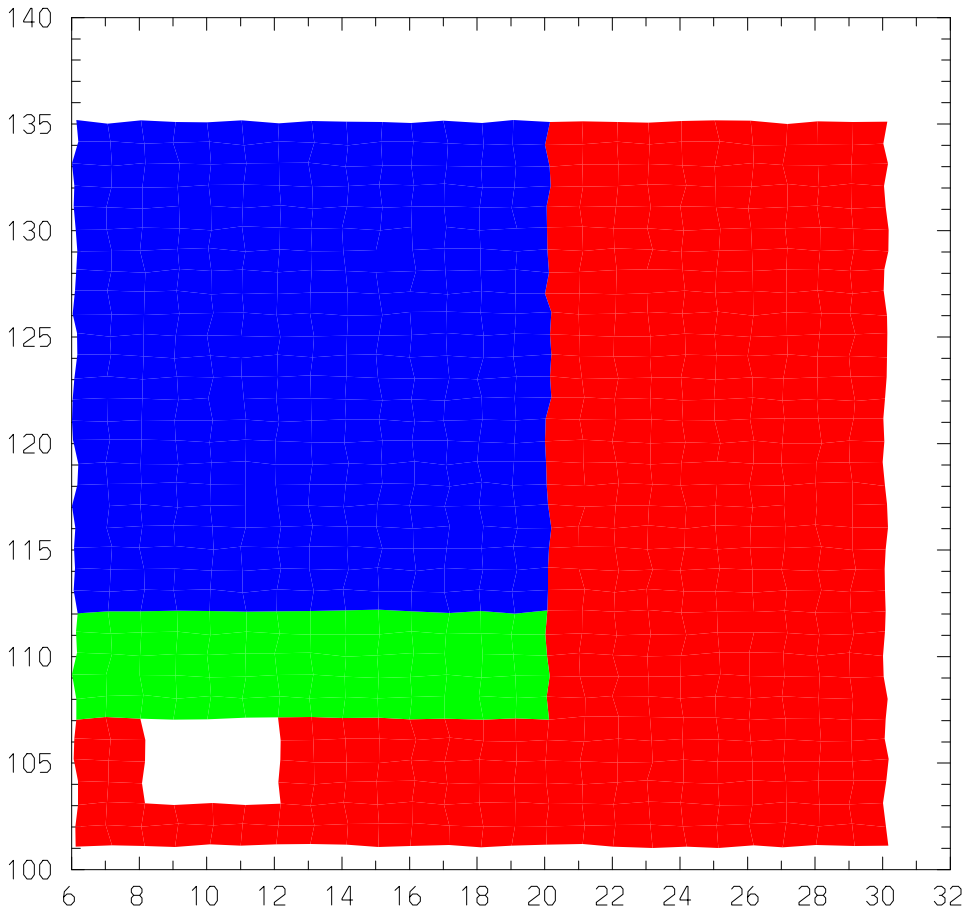
The

`plotpf` command plots a color-filled mesh which displays the physics quantity *pvar* on the polygonal mesh specified by *x,y* with colors. If specified, *¡keylist>* is a list of optional keywords and values. This command is analogous to the use of `plotf` for a (k,l)-mesh.

The colors assigned to the individual zones range from the beginning color in the colormap (after the "*named colors*" *red, green, blue, yellow*, etc.) to the last color in the colormap. The color varies from low color index to high color index as *pvar* varies from its minimum to maximum values.

The mapping of colors can be *linear*, *logarithmic*, or *normally distributed*. Use the attribute cscale to specify the mapping choice. 49.3See "plotf: Fillmesh Plot" on page 57 for the available options. A colored annotation on the right side of the frame displays the assignment of colors to the corresponding values of *pvar*.

The attribute `zlim=[`*zmin*`,`*zmax*`]` allows the user to specify limits to be used when mapping physical values to colors (by any of the above-mentioned color scales). If not supplied, the minimum and maximum values in *pvar* are used. The use of `zlim` allows one to use the same colormap for a series of related plots, such as the time-evolution of *pvar*.




```
Region 1 is colored red
Region 2 is colored green
Region 3 is colored blue
```

Figure 50.2: Use of plotp to color region map

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified, i.e. they are not remembered across commands.

`grid`, `cscale`, `legend`, `zlim`

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

The same annotation is given to the right of the plot as for `plotf`. (See 49.3.1“Fillmesh Level Annotation” on page 58 for control over its appearance.)

CHAPTER

FIFTYONE

Surface Plot Commands

EZN contains a limited number of capabilities for producing wire-frame plots of surfaces defined by data on a rectangular mesh. These are *not* true EZN functions in that they do not add the plots to the EZN display list.

51.1 srfplot: 3-D Surface Plot

Calling Sequence

```
srfplot(x, y, z, nx, ny, view)
```

Description

The `srfplot` call is used to generate a 3-D surface (wire-frame mesh) plot of `z` versus `x` and `y`. In particular, `z` is a matrix of values of size `nx` by `ny`, where `x` and `y` are vectors of length `nx` and `ny`, respectively. The function plotted is then `z(i,j)` = *fcn*(`x(i),y(j)`). The viewpoint of the plot is given by the two-vector `view` where `view(1)` is the angle from the x-axis in the xy-plane and `view(2)` is the angle from the xy-plane. (Angles are in degrees.) Various parameters can be set to control the labels and presentation of the surface plot; see the following subsection.

The `srfplot` subroutine calls the NCAR Graphics routine `SRFACE` and is therefore limited in its interaction with the rest of EZN graphics. In particular, a surface plot cannot share the frame with any other plot, although text may appear. A surface plot cannot be mapped to a quadrant.

Note: The `srfplot` routine is *not* a true Basis Function; it doesn't add the plots to the EZN display list. Commands like "`cgm send`" do not work; you must first activate the desired plotting device(s) before calling `srfplot`.

External Parameters

A number of options to the `srfplot` routine may be controlled through external parameters. These are detailed below.

- Plot Limits – There are 6 external parameters to control plot limits. These are `srfxlo`, `srfxhi`, `srfylo`, `srfyhi`, `srfzlo`, and `srfzhi`. These parameters are used to specify the minimum and maximum of the x-, y-, and z-data. In particular, for a series of surface

plots, these may be set to values and then left "frozen" so that plot comparisons can be made. If the value of the `srfautoscal` parameter is set to `false` then the values of the six limit parameters are used to determine how the plot is scaled. If `srfautoscal` is `true` then the plot is automatically scaled to fill the frame and the limit parameters are ignored. The default is to perform automatic scaling.

- Plot Labels – The parameters `srfxtle`, `srfytle`, and `srfztle` can be set to put titles on the axes. These character strings (maximum length 80) are also used in the legend. If axis labels are not desired, then the parameter `srflabel` can be set to `false`. The legend will still use the axis label parameters regardless of the setting of `srflabel`. The default is to have axis labels.

- Plot Title – The parameter `srftitle` is a character string (maximum length 80) which is used to label the plot, analogous to the super-title for other EZN plots.

- Legend Location – The legend may be located either in the upper right-hand corner of the plot, or the lower right-hand corner of the plot. The default is to put the legend at the top, but this can be overridden by setting `srftopln` to `false`. Note that a relatively long title can intrude into the legend when the legend is located at the top of the plot.

- Skirt – If `srfiskrt` is `true` a "skirt" is plotted around the base of the surface. The height of the skirt can be controlled by parameter `srfhskrt`. The default is no skirt.

- Plot Resolution – For very large data sets, the number of points to be plotted in the x- and y-directions can be specified via the `srfnpx` and `srfnpy` parameters. The default is 100 points in each direction. If `nx`>`srfnpx` or `ny`>`srfnpy`, the data are thinned to the resolution specified by these parameters. Requesting too much resolution can produce a very dense plot where details are obscured.

Example

```
integer n=20, i, j
real r, view(2) = [60., 60.]
## Legends and labels for axes:
srfxtle="X Axis"
srfytle="Y Axis"
srfztle="Z Axis"
srftitle="Sombrero Function"
srftopln=false   ## Put legend at the bottom instead of the top.
real x(-n:n) = iota(-n,n)
real y(-n:n) = iota(-n,n)
real z(-n:n, -n:n)
do i = -n, n
   do j = -n, n
      r = sqrt(x(i)**2 + y(j)**2) + 1e-6
      z(i,j) = sin(r) / r
```

```
   enddo
enddo
srfplot(x, y, z, 2*n+1, 2*n+1, view)
```

51.2 isoplot: 3-D Isosurface Plot

Calling Sequence

```
isoplot(t, nx, ny, nz, c0, view)
```

Description

The `isoplot` call is used to generate a 3-D surface (wire-frame mesh) approximation to the isosurface *fcn(x,y,z)*=`c0`, where `t` is a three-dimensional array of size `nx` by `ny` by `nz` containing values of *fcn* on a (uniform) rectangular mesh. The viewpoint of the plot is given by the two-vector `view` where `view (1)` is the angle from the x-axis in the xy-plane and `view (2)` is the angle from the xy-plane. (Angles are in degrees.) Various parameters can be set to control the labels and presentation of the isosurface plot; see the following subsection.

The `isoplot` subroutine calls the NCAR Graphics routine ISOSRF and is therefore limited in its interaction with the rest of EZN graphics. In particular, an isosurface plot cannot share the frame with any other plot, although text may appear. An isosurface plot cannot be mapped to a quadrant.

Note: The `isoplot` routine is *not* a true Basis Function; it doesn't add the plots to the EZN display list. Commands like "`cgm send`" to not work; you must first activate the desired plotting device(s) before calling `isoplot`.

External Parameters

A number of options to the `isoplot` routine may be controlled through external parameters. These are detailed below.

- Plot Controls – parameter `isoflg` serves two purposes.

- First, the absolute value of `isoflg` determines which types of lines are drawn to approximate the surface. Three types of lines are considered: lines of constant x, lines of constant y, and lines of constant z. The following table lists the types of lines drawn:

- Plot lines of constant

- `abs(isoflg)` x y z

- 1 no no yes

- 2 no yes no

- 3 no yes yes

- 4 yes no no
- 5 yes no yes
- 6 yes yes no
- 0, 7 or more yes yes yes
- Second, the sign of `isoflg` determines what is inside and what is outside, hence which lines are visible and what is done at the boundary of the data. For `isoflg`>0, `t` values greater than `c0` are assumed to be inside the solid formed by the drawn surface. For `isoflg`¡0, `t` values less than `c0` are assumed to be inside. If the algorithm draws a cube, reverse the sign of `isoflg`.
- The default value is `isoflg`=7 (plot lines of constant x, y, and z).

- Plot Labels – The parameters `isoxtle` and `isoytle` can be set to put titles on the axes. If axis labels are not desired, then the parameter `isolabel` can be set to `false`. The default is to have axis labels.
- Plot Title – The parameter `isotitle` is a character string (maximum length 80) which is used to label the plot, analogous to the super-title for other EZN plots.
- Plot Resolution – For very large data sets, the number of points to be plotted in the x- and y-directions can be specified via the `isonpx`, `isonpy` and `isonpz` parameters. The default is 100 points in each direction. If `nx`>`iosnpx`, `ny`>`iosnpy` or `nz`>`iosnpz`, the data are thinned to the resolution specified by these parameters. Requesting too much resolution can produce a very dense plot where details are obscured.

Example

The following example generates a picture of the 3-D unit ball. The value of `c0` is 0.5 here. Note that the triple loop takes a long time to execute.

```
isotitle=''Unit ball in R3''
isoxtle=''x''
isoytle=''y''
integer nx=10, ny=10, nz=10
real t(-nx:nx,-ny:ny,-nz:nz)
integer i, j, k
real x, y, z
do k = -nz, nz
   z = k / (1.0*nz)
   do j = -ny, ny
      y = j / (1.0*ny)
      do i = -nx, nx
         x = i / (1.0*nx)
```

```
         t(i,j,k) = x*x + y*y + z*z
      enddo
   enddo
enddo
isoflg=-7 ## Tell ISOSRF that values < 0.5 are inside.
isoplot(t, 2*nx+1, 2*ny+1, 2*nz+1, 0.5, [15.,15.])
```

-

CHAPTER

FIFTYTWO

Frame Control

There are four commands which control frame actions. The `frame` command sets the limits of the picture frame. The `nf` (New Frame) command is used to begin a new frame. The `sf` (Show Frame) command is used to display the current frame to all active devices. The `undo` command removes a plot command previously issued in a frame.

52.1 frame: Set Frame Limits

Calling Sequence

```
frame
\textit{xmin,xmax,ymin,ymax}fr
\textit{xmin,xmax,ymin,ymax}
```

Description

The **frame** command sets the limits of the picture frame, which are *frame* type attributes. The `frame` command applies immediately to all plot commands in the frame. `fr` is an abbreviation for "`nf; frame`".

You can supply zero to four arguments. If specified, *xmin* is the minimum value for the x scale, *xmax* is the maximum value for the x scale, *ymin* is the minimum value for the y scale, and *ymax* is the maximum value for the y scale. For each value not specified, the extreme value of the data will be used to calculate the limit. In this case, skipped arguments should be indicated by commas. (Don't put a comma after the last argument you are supplying: Basis' line continuation convention will bite you.) The frame limits will not be retained across frame advances. If a frame already contains objects it will be displayed with these frame limits.

Control Variables and NCAR Autograph Parameters

Some EZN control variables and NCAR Autograph parameters can be used to fine tune the limits of a frame. For example, ezcextra controls the extra space below and above the *ymin* and *ymax* when the frame limits are determined by the data extrema. Set ezcextra=0. to get rid of this extra space.

The NCAR Autograph package will extend the axes to accommodate labels for the last major ticks in the default case. The parameters "X/NICE.", "Y/NICE." in Autograph can be set to 0 to disable this default behavior. EZN makes agseti, agsetf, agsetc, aggeti, aggetf, and aggetc visible to the user for interactively fine tuning the graphics. For reference, 53See CHAPTER 11: "Axes, Titles and Text" on page 85, 57See CHAPTER 15: "Control Variables and Defaults" on page 101, and NCAR Autograph documents for details.

Examples

In the first example, the frame limits are set to the specified values. In the second example, the extreme values for *xmin* and *ymin* are used. Hence, the frame limits are 1,5,1,9.

```
ezcshow=true
plot iota(10),iota(10)
frame 2,9,3,7
frame ,5,,9                # xmin,ymin defaulted
frame 2,9                  # ymin,ymax defaulted
```

Since `ezcshow` is `true`, four frames are displayed, as illustrated on the following pages. If `ezcshow` had been set `false`, only three frames would be displayed. The moral is: put the `frame` command first, normally, and use subsequent `frame` commands to plot different views of the same set of objects.

After a picture is displayed, the four values actually used as frame limits are available in the variables xminu, xmaxu, yminu, ymaxu. These can be used in calculating arguments to subsequent `frame` commands. In contrast, variables xmin, xmax, ymin, ymax will contain the most recent arguments supplied to frame. As an exercise, repeat the above example but type

```
xmin,xmax,ymin,ymax
xminu,xmaxu,yminu,ymaxu
```

after each plot command. Note that only the `xmin`,`xmax`,`ymin`,`ymax` values actually given in the previous `frame` command change.

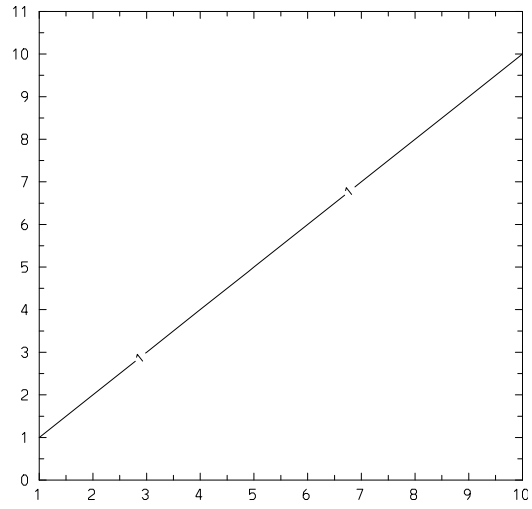
52.2 nf: New Frame

Calling Sequence

```
nf
```

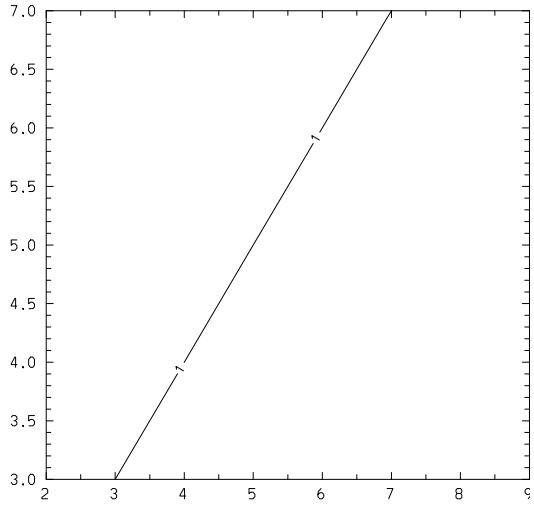
Description

The **nf** command signals that a new frame is to be started. By default, attributes set by the `attr` command are reset to their default values when a new frame is issued.



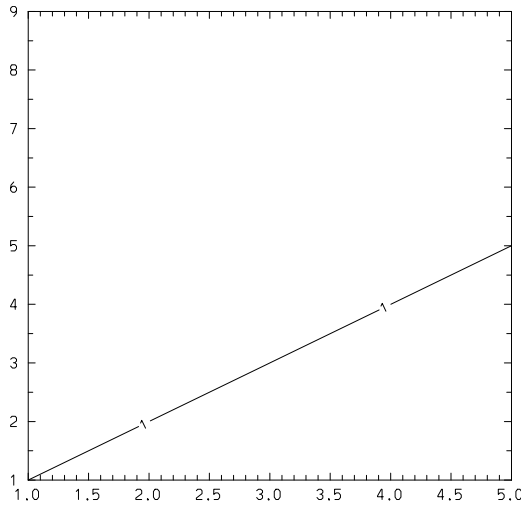

```
1: plot iota(10) iota(10)
```

Figure 52.1: Example of frame setting: default



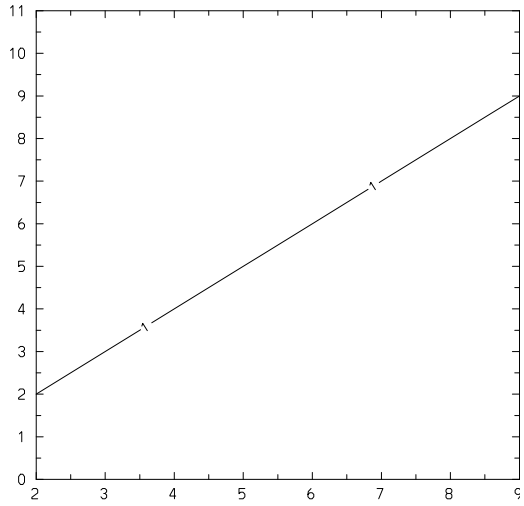

```
1: plot iota(10) iota(10)
```

Figure 52.2: Example of frame setting: frame 2,9,3,7




```
1: plot iota(10) iota(10)
```

Figure 52.3: Example of frame setting: frame ,5,,9




```
1: plot iota(10) iota(10)
```

Figure 52.4: Example of frame setting: frame 2,9

If variable ezcreset is set to false, then the attributes set by the `attr` command remain in effect across frame advances.

What `nf` really does is to close the currently displayed frame. If you are using windows, the effect of `nf` depends on whether or not `ezcshow` is true or false.

- If `ezcshow` is true, you are already looking at the picture, and a `nf` will clear the screen to begin the next one.
- If `ezcshow` is false, you haven't seen the picture yet so `nf` displays it, and this frame will remain displayed, until the next `nf`.

An automatic `nf` is done when the program ends to finish the last frame if required.

Examples

In the default case, the line style is reset across frame advances.

```
ezcreset=true      # (default)
attr style=dashed
plot y,x            # First plot dashed.
plot y2,x2          # Second plot dashed.
nf
plot y3,x3          # Style IS reset to solid (default).
```

In the example below, the line style remains dashed across frame advances.

```
ezcreset=false
attr style=dashed
plot y,x            # First plot dashed.
plot y2,x2          # Second plot dashed.
nf
plot y3,x3          # Style NOT reset across frame advance.
```

(A better way to do this is usually to change the default variables, in this case defstyle.*)*

52.3 sf: Show Frame

Calling Sequence

```
sf
```

Description

The **sf** command displays the current frame to all active devices. The frame is displayed regardless of the value of variable `ezcshow`. This command is useful when a user wants to control the display of the frame at certain times; i.e., not every time a graphic object is added on a frame (default).

Examples

In the example below, the `sf` command is used to display the frame after 3 curves have been added. Note that variable ezcshow was set to false. A fourth curve can then be added; had `nf` been used instead of `sf`, the first three curves would no longer be in the picture.

```
ezcshow=false
plot y1,x1
plot y2,x2
plot y3,x3
sf          # Force show of current frame.
plot y4,x4
nf
```

52.4 undo: Undo a Plot Command

Calling Sequence

```
undo
\textit{number}
```

Description

Remove the *number*'th object in the EZN display list. (Each frame EZN has a list of graphic objects when display is requested.) If no argument is given, `undo` the last graphic object. Some EZN commands do not generate graphic objects in the display list (for example, the `frame` command), so *cannot be undone* in this way. The easist way is to use the legend as a reference list for `undo`. For graphic objects whose *legend* have been *suppressed*, it is the user's responsibility to figure out which number should be supplied for `undo`. -

CHAPTER

FIFTYTHREE

Axes, Titles and Text

The axes of a frame are drawn by the NCAR Autograph package. A set of parameters can be set by the user to fine tune the settings of the axes.

There are different ways to plot titles and informational text on a frame. Several variables can be used to control the size, appearance and scope of setting of the titles and text.

53.1 Changing Autograph Parameters

NCAR graphics packages have many parameters which control the appearance of the pictures. One sets these parameters by calling a routine with the name of the parameter and the value. Typically there are three routines for each package used to set real, integer, and character values.

In the NCAR Autograph package, the routines `agsetf("name",fval)`,

`agseti("name",ival)`, and `agsetc ("name","string")` are used to set the parameters. Note that the "set" call must be made *before* the plot command(s) it is to modify. Use `aggetf`, `aggeti`, or `aggetc` to determine the current setting for a variable. For example, the commands `integer ixnice; call aggeti("X/NICE.",&ixnice)` will return the current value of Autograph parameter `X/NICE.` in Basis variable `ixnice`.

In drawing pictures, EZN makes calls to set the following variables or groups of variables, so they *cannot be set by the user*.

```
WINDOW.
GRAPH.
X.MINIMUM.
X.MAXIMUM.
X.LOGARITHMIC.
Y.MINIMUM.
Y.MAXIMUM.
Y.LOGARITHMIC.
GRID.
BAC.
AXIS/s/CONTROL.
```

```
AXIS/s/TICKS/MAJOR/LENGTH/INWARD.
AXIS/s/TICKS/MINOR/LENGTH/INWARD.
LABEL/CONTROL.
DASH/SELECTOR.
DASH/LENGTH.
DASH/PATTERNS/1.
```

In the above, `s`=`LEFT`, `RIGHT`, `TOP`, or `BOTTOM`.

The following are the Autograph parameters a user is most likely want to change (note the final period "`.`" is part of the name). A complete list of parameters for controlling axes is given in the NCAR Autograph document (available on the web at http://ngwww.ucar.edu/ngdoc/ng/supplements/autograph/).

Name	**default**	**other settings**
`X/NICE.`	-1	0 disables "nice" x-axis.
`Y/NICE.`	-1	0 disables "nice" y-axis.
`AXIS/s/TICKS/MAJOR/COUNT.`	6	n>0*:* use n+2 to 5n/2+4 major tick marks on linear axes.
`AXIS/s/TICKS/MINOR/SPACING.`	Autograph chooses	n¡1: no minor tick marks. n>=1: n minor tick marks per major tick mark.
`AXIS/s/NUMERIC/TYPE.` (See Autograph documentation for details.)	Autograph chooses the format	0: no numeric labels. 1: scientific notation. 2: exponential notation. 3: "no exponent" notation.
`AXIS/s/NUMERIC/EXPONENT.`	Autograph chooses	See Autograph documentation. (Used with `TYPE.`)
`AXIS/s/NUMERIC/FRACTION.`	Autograph chooses	See Autograph documentation. (Used with `TYPE.`)
`s=LEFT,RIGHT,TOP,BOTTOM`		

Example

For example, you may call `agseti("AXIS/BOTTOM/TICKS/MAJOR/COUNT.",3)` to reduce the number of major tick marks on the bottom axis from the default 8-19 to 5-11, and call `agseti("AXIS/BOTTOM/TICKS/MINOR/SPACING.",4)` to introduce four minor tick marks per major tick mark.

53.2 titles: Put Titles on a Plot

Calling Sequence

```
titles
\textit{top, bottom, left, right}
```

Description

Put up to four quoted strings (up to 120 characters each) at the top, bottom, left, and right of the picture, respectively. Each title can also be set individually by assigning a quoted string to the variables titlet, titleb,

titlel, and titler.

The default value of each title is a blank string. These titles are cleared by `nf`.

The variable ezctitle can also be set to a string. This string, referred to as the *supertitle*, will appear on every subsequent frame, usually at the top. The control variable ezcsuper can be set false to place it at the bottom.

The variables ezctitfr controls the size of the titles relative to the graph; the default value is to use 4% of the picture for each title, with 60% of ezctitfr used for the supertitle.

For further information, 46.2see Section 4.2 ”Controlling Layout” on page 23.

53.3 text: Put Text in the Interior of a Plot

Calling Sequence

```
text
\textit{string, x, y, nsize, angle, center}
```

Description

Write the *string* (up to 120 characters) on the plot beginning at coordinates *x*,*y* in *User’s World Coordinates*, using a size argument to the Autograph routine `agpwrt` of *nsize*, at *angle* degrees to the x-axis. The centering of the text with respect to the point (*x*,*y*) is done by passing *center* to `agpwrt`. Usual values for *center* are -1, the default, which centers the left edge at (*x*,*y*); 0, which places the center of the string at (*x*,*y*); and 1, which centers the right edge at (*x*,*y*).

The arguments *nsize*, *angle*, and *center* can be omitted. The defaults are to use text of a minimum size, horizontal, and with the left center edge of the text at the point (*x*,*y*). The text will not be smaller than the size specified by the last `ezcminsz` call (see 46.2page 23).

Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified; i.e., they are not remembered across commands.

grid, **scale**, **font**, **color**, **legend**

If optional attributes are given on the plot command line, they are specified in the usual form:

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

Although it is recognized, the `font` attribute currently has no effect. (For font control, 53.5see Section 11.5 ”Text Quality and Optional Fonts” on page 90.)

Example

53.4 ftext: Put Text Anywhere in a Frame

Calling Sequence

```
ftext
\textit{string, x, y, nsize, angle, center}
```

Description

Write the *string* on the frame beginning at coordinates *x,y* in *Normalized Device Coordinates* (i.e. the whole frame is a [0.,1.]x[0.,1.] unit square), using a size argument to the Autograph routine `agpwrt` of *nsize*, at *angle* degrees to the x-axis. The centering of the text with respect to the point (*x,y*) is done by passing *center* to `agpwrt`. Usual values for *center* are -1, the default, which centers the left edge at (*x,y*); 0, which places the center of the string at (*x,y*); and 1, which centers the right edge at (*x,y*).

The arguments *nsize*, *angle*, and *center* can be omitted. The defaults are to use text of a minimum size, horizontal, and with the left center edge of the text at the point (*x,y*). The text will not be smaller than the size specified by the last `ezcminsz` call (see 46.2page 23).

The differences between the text and ftext commands are:

- The coordinate system: the text command uses *User’s World Coordinates* and the ftext uses *Normalized Device Coordinates*.
- A string specified with User’s World Coordinates will be relocated relative to the frame limits, but a string specified with Normalized Device Coordinates is anchored at the given location on the frame regardless of frame limits changes. Furthermore, a string specified with User’s World Coordinates will be clipped to the frame limits.

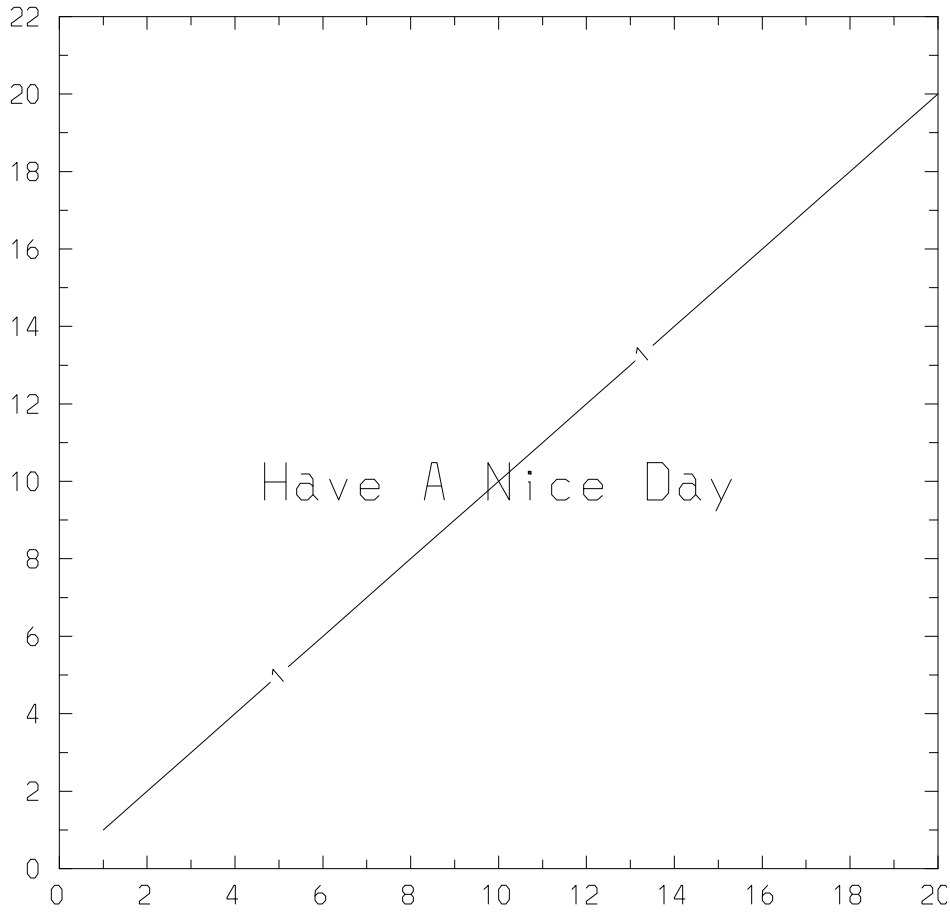
Optional Attributes

The following optional attributes can be specified with this command. For *object* attributes, they are local to the command specified; i.e., they are not remembered across commands.

font, **color**, **legend**

If optional attributes are given on the plot command line, they are specified in the usual form:

```
# Example of text command
plot iota(20)
text "Have a Nice Day" 10 10 24 0 0
nf
```




```
1: plot iota(20)
text "Have A Nice Day" 10 10 24 0 0
```

Figure 53.1: Example of adding text

key1=value1,key2=value2,...,keyN=valueN

To set an *object* attribute across commands use the `attr` command. 47.3See “Attribute Table” on page 31 for descriptions of the values which can be assigned to these keywords.

Although it is recognized, the `font` attribute currently has no effect. (For font control, 53.5see Section 11.5 ”Text Quality and Optional Fonts” on page 90.)

53.5 Text Quality and Optional Fonts

On some occasions you may want to use different fonts and/or different quality of the text. EZN provides a routine ezcstxqu(intq) for user to change text quality. The input parameter intq is an integer with possible values *0 for high quality, 1 for medium quality (default), and 2 for low quality*. Higher quality of the text requires more computer resources.

The text quality affects the appearance of *labels on the axes*, *titles*, and the *text strings* specified by the text and the ftext commands. It currently has no effect on the appearance of the level annotations for contour or fillmesh plots.

Only *high quality text* (intq=`0`) will allow one to use different fonts; for example, the Greek letters. The optional fonts that came with the NCAR distribution need to be installed on the computer system on which EZN is running; check with your System Manager for the availability of these fonts. In order to specify different fonts by using ASCII characters, the user needs to use function codes embedded in the text string to indicate special letters or symbols. When high quality text is being used, the character size *nsize* is free from the restriction set by ezcminsz. Refer to the NCAR Plotchar user’s manual for the details. -

CHAPTER

FIFTYFOUR

Stream Output to Graphics

Basis contains a variable stdplot which can be used as the unit number in stream output statements. Basis also can be told to redirect most of its output to the graphics package with the "`output graphics`" command. Both of these commands result in calls to a primitive routine `ptext` which writes a line onto the graphics page. The interaction with EZN is as follows:

1. The first such output line will begin on a new frame;
2. Subsequent lines will appear below the previous lines until the frame is full;
3. The number of lines which will fit on a frame is controllable by setting variable `nptext`. The default is 45 lines/frame.
4. The next line to write on can be changed by setting variable `textline`.
5. A `nf` command or an EZN plot command will start a new frame.

Example

As an example, the following puts the Basis version message on the frame together with a message showing the value of a variable and a group named InputStuff:

```
echo=no
output graphics     #Redirect to graphics device.
version         #Print Basis version message.
stdplot <<return    #Print blank line.
stdplot << "This run with alpha = " << alpha
InputStuff
output tty      #Redirect back to tty.

-
```

CHAPTER

FIFTYFIVE

Quadrant Mode

You can use the EZN package in *quadrant mode* to place several different pictures on the same frame, and to mix text output with pictures. The routine `ezcsquad` provides general control, while the easy-to-use `ezcquad` routine allows you to put up to four different EZN stream output sessions on the same frame.

`ezcsquad(xmin,xmax,ymin,ymax)` sets the portion of the frame into which the plotting will occur. The four arguments must be in [0.,1.] coordinates. The EZN package is put into quadrant mode, as described below.

`ezcquad(iquad)` is an easier-to-use facility built upon `ezcsquad`. The quadrants are numbered:

```
1  2
3  4
```

The join of 1 and 2 is called 12, the join of 1 and 3 is 13, and likewise for 34 and 24, and finally 1234 is the usual full frame. The command `ezcquad(iquad)` where `iquad` is 1, 2, 3, 4, 12, 13, 24, 34, or 1234, sets the quadrant accordingly by calling `ezcsquad` with appropriate arguments. These appropriate arguments are calculated with respect to a default full frame, but you can call `ezcdquad (xmin,xmax,ymin,ymax)` to set a portion of the screen as the default frame, which is in turn chopped up into quadrants by `ezcquad`.

In quadrant mode, EZN tries to scale everything appropriately. It is usually wise not to use large values for the minimum text size. The number of lines per page for `ptext` output is scaled by the percentage of the vertical space the quadrant occupies. The frame does not advance if too many lines are written; rather, writing returns to the top of the quadrant.

When a subsequent `ezcquad` call is issued, it is as if a `nf` were issued except the frame is not actually advanced; rather, the next plot commands will make a picture in the new quadrant. The delayed-display mode `ezcshow=false` is turned on in quadrant mode.

The `sf` *command should not be used in quadrant mode.*

When the last quadrant is completed, `nf` starts a new frame, and puts you back into non-quadrant mode with a full-screen picture (as defined by the last call to `ezcdquad`). (Note that `ezcshow` is still set to FALSE.)

Examples

The following example plots four graphs on one frame:

The following example puts a long skinny plot on the top, and some text underneath, with no legend. We turn the echo off so as not to echo the commands themselves in the picture.

Note that the quadrant mode does not interact correctly with the `send` device commands. You will have to use manual control of the output to each device in order to single out a frame for sending to a hardcopy device. Generally, we expect quadrant mode to be used for production output rather than output from interactive exploration.

```
integer i
real x=iota(-5,5)
do i=1, 4
   ezcquad(i)
   plot x**i legend="plot x**"//format(i,0)
enddo
nf
```

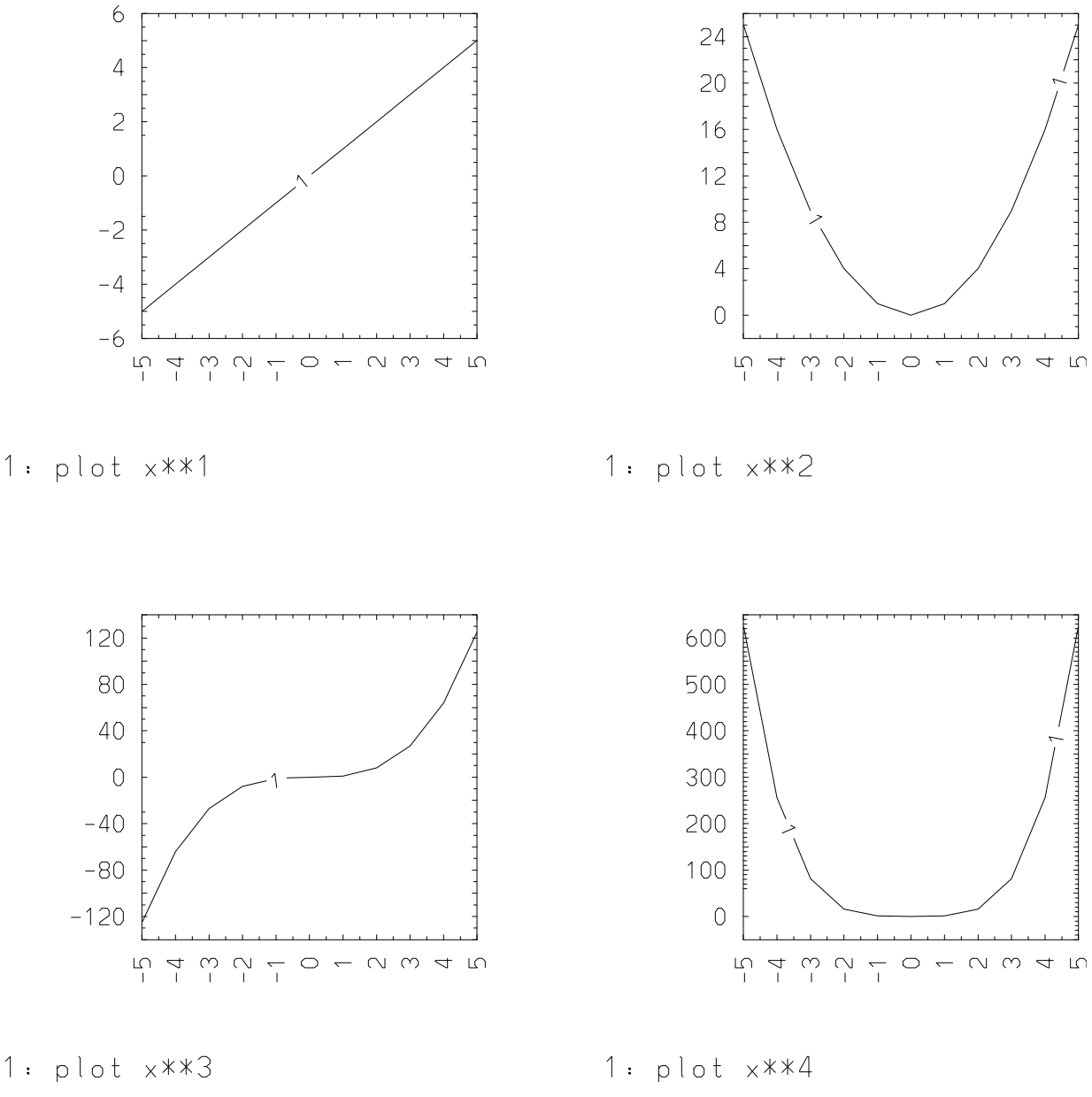
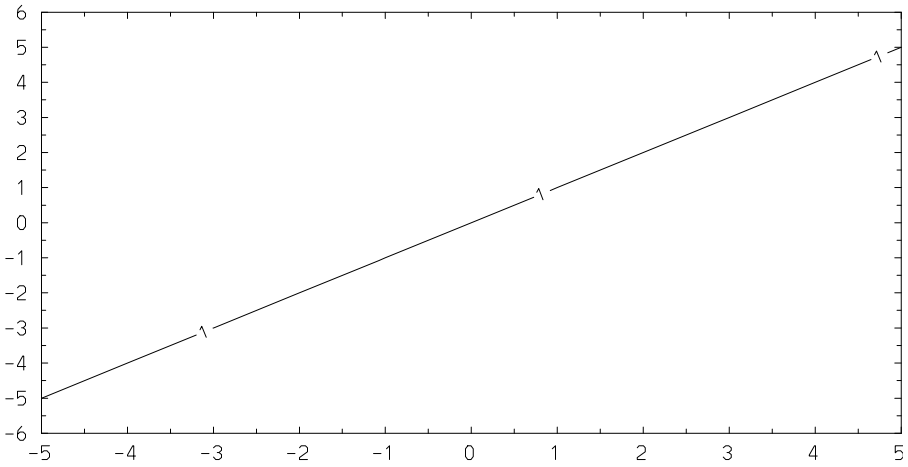


Figure 55.1: Example of multiple quadrant plot

```
echo = no
ezcfixed = no
integer i
real x=iota(-5,5)
ezcquad(12)
plot x legend=no
ezcquad(34)
output graphics
nptext = 22 #Want 11 lines to fit exactly.
do i=-5,5
   x(i)
enddo
nf
output tty
```




```
x(i)       =    -5.00000E+00
x(i)       =    -4.00000E+00
x(i)       =    -3.00000E+00
x(i)       =    -2.00000E+00
x(i)       =    -1.00000E+00
x(i)       =     0.00000E+00
x(i)       =     1.00000E+00
x(i)       =     2.00000E+00
x(i)       =     3.00000E+00
x(i)       =     4.00000E+00
x(i)       =     5.00000E+00
```

Figure 55.2: Example of output graphics

CHAPTER

FIFTYSIX

Interactive Graphics Tools

EZN has several interactive graphics tools that are available for general graphics applications and/or for Lasnex-specific applications. The interactive tools use point and click to interact with the graphics display and to probe the physics quantities. They are applicable only when an X-window is currently open and has an EZN-generated plot displayed on it.

The commands described in this chapter are not part of EZN *per se*. EZN/EZD provides the "hooks" that are necessary to implement them via functions `ezdprobe` and `ezczoom`.

These commands are defined in utility file `interactive.in`, which is generally installed in $BASIS_ROOT/include. To use these commands from Basis or a Basis application, one must first execute command "`read interactive.in`". (This is not necessary for users of Sod or Lasnex, because this file is read at code initialization.)

Once the definitions have been read, the command "`help graphics`" will display a summary of the available interactive graphics commands at the terminal.

56.1 General Graphics Applications

56.1.1 Zoom

The `zoom` function enlarges the picture in a rectangular region bounded by two mouse clicks at the diagonal points. The contents of the graphics display within the selected region will be redrawn to fill the whole frame.

The `zoom` command has the advantage over `frame` with arguments of making it easier to select a specific region of interest. The user will be able to examine the details of the picture by using `zoom` to "`zoom` in" repeatedly.

56.1.2 Unzoom

The `unzoom` command will return the frame to the previous stage, which gives a chance to select another portion of the frame for further zoom in. The user will be informed when `unzoom` has returned to the original frame. Subsequent `unzoom` commands will have no effect.

A `frame` command without arguments can be used to return the picture to its original size as a short-cut to a series of `unzoom` commands.

56.2 Lasnex-Specific Applications

A set of Lasnex-specific interactive tools can be invoked to mark nodes, to mark a special k-line or l-line, or to highlight a region. One can also request the id of selected zones.

All of these commands assume that a window is open and a mesh-oriented command (see 49CHAPTER 7: "Mesh-Oriented Commands") has been executed to display a picture in it. They can be used to find information about the mesh associated with the plot.

Caution: These commands require the standard Lasnex mesh variable names to be used. Thus, the mesh must be dimensioned `kmax` by `lmax`, with horizontal axis variable `zt`, vertical axis variable `rt`, and region map `ireg`. The variables `ezdx`, `ezcy`, `ezcireg` are ignored.

A command with a repeated last letter is used to mark a series of points, lines, segments, regions, or zones. Click outside the frame to end the command.

The appearance of markers or highlights can be controlled the same way as the user's plot commands. If the variable *ezcshow* is *true*, then results will be shown immediately; if *ezcshow* has been set to *false*, then they will not be shown until either *sf* or *nf*.

56.2.1 Marking Points

`markp` and `markpp` are used to mark the node(s) pointed to by the mouse. Within the zone where the mouse was clicked, the closest node will be indicated by a circle and its node indices (K,L) will be displayed at the terminal.

The color of the displayed marker(s) can be controlled by preceding the command with "`attr color=`*mycolor*".

56.2.2 Marking Mesh Lines

`markl` and `markll` are the commands to mark one or more k- or l-lines. The user uses the mouse to click two distinct points on a k-line (or an l-line). Then the k-line (or the l-line) containing these two clicked points will be highlighted. As with `markp`, the indices of the node nearest each click will be displayed at the terminal. If the two nodes are not on the same k- or l-line, an error message is given and nothing is plotted.

The color of the highlighted line(s) can be controlled by preceding the command with "`attr color=`*mycolor*" or "`attr kcolor=`*mykcolor* `lcolor=`*mylcolor*". Note: `kcolor` or `lcolor` takes precedence over `color` when all are set and have values other than `fgcolor`.

56.2.3 Marking Mesh Segments

`marks` and `markss` are the commands to mark segments of k- or l-lines. It is used exactly the same as `markl`, but only the portion of the line between the two clicked points will be highlighted.

The color of the highlighted segment(s) can be controlled by preceding the command with an appropriate `attr` command, as for `markl`.

56.2.4 Marking Regions

`markr` and `markrr` are the commands to mark regions bounded by two nodes not on the same k- or l-line. The mesh in the region(s) will be highlighted.

Note: `markr` is actually a function that returns the limits of the region, (*kmin*, *lmin*, *kmax*, *lmax*). To eliminate the extraneous output, use it as "`call markr`".

The color of the highlighted mesh can be controlled by preceding the command with an appropriate `attr` command, as for `markl`.

56.2.5 Marking Zones

`markz` and `markzz` are the commands to mark nodes. When a point is clicked, the zone containing it will be highlighted and its zone indices (K,L) will be displayed at the terminal.

If the variable `verbose=yes`, the (r,z)-coordinates of the clicked point(s) will be displayed along with the zone indices.

The color of the highlighted zone(s) can be controlled by preceding the command with an appropriate `attr` command, as for `markl`. -

CHAPTER

FIFTYSEVEN

Control Variables and Defaults

This chapter discusses various variables (also called *parameters*) that are available to control the details of the behavior of the EZN package, as well as routines to query and set parameters.

57.1 EZN Control Variables

There are two groups of variables in the EZN package which control the details of its behavior. The first group, Ezcurve, controls most of the details of the display. The second, EzcurveDefaults, controls the default values of the attributes. The best way to get up-to-date documentation on these is to use the `list` command:

```
output graphdoc
list
Ezcurve,
EzcurveDefaults
output tty
```

and then print the file `graphdoc`.

57.1.1 Ezcurve Variables

Here are details on many of the variables in group Ezcurve.

`ezcnoplot` If `true`, enter no-plot mode, which makes all graphics commands no-ops. Intended for use in parallel applications in which only the "master" node generates plots. Default: `false`.

ezcshow Determines if the current picture is displayed each time it is changed by an EZN command, or only when a *frame* attribute is changed or an `nf` command is issued. By default, `ezcshow=true`.

ezcreset Determines if attributes set with the `attr` command are reset to the default values upon a frame advance. If `ezcreset=false`, attributes will remain set across frame advances. By default, `ezcreset=true`.

ezcextra If frame limits in the y-direction are not specified, the limits of the data are used, but a small extra space is left on the top and the bottom; this variable contains the factor for the amount of such space (default: .025). Set this variable to 0. to eliminate the space.

ezcnocx This logical variable determines whether or not to allow EZN to average an x array that is one too long to plot a y array against it. Default: ezcnocx=`false`, which means that x can be changed.

ezcnocy This logical variable determines whether or not to allow EZN to average a y array that is one too long to plot against the x array. Default: ezcnocy=`false`, which means that y can be changed.

`ezclbshft` Control amount to shift labels for subsequent curve or ray plots to avoid overplotting. An integer in range 0-3. Default: 0.

ezcfloor The minimum value for log plots is ezcfloor times the maximum data value. The default value is 0., which means to use `r1mach(3)`, approximately 1.E-7.

`dflogstyle` If `true`, log plots have line style `solid` for data above the floor and `dotted` for data which have been promoted to `ezcfloor`. Default: false.

ezctitle String valued, sets the *supertitle*, put on for every frame; not cleared by nf. Default: a blank string.

ezcsuper The supertitle is at the top if `true`, at the bottom if `false`. Default: `true`.

titlet String valued, sets the *top title* for a frame. Default: a blank string.

titleb String valued, sets the *bottom title* for a frame. Default: a blank string.

titlel String valued, sets the *left title* for a frame. Default: a blank string.

titler String valued, sets the *right title* for a frame. Default: a blank string.

ezcfixed If `true`, the plot box will always be the same size regardless of titles and contour labels. If `false`, the plot box expands to its limit. Default: `true`.

ezctitfr Fraction of the frame to devote to titles. 0.6 of this is used for ezctitle. Default: 0.04.

ezclegfr Fraction of the frame to devote to the legend. Default: 0.125.

ezccntfr Fraction of the frame to devote to the contour level list. Default: 0.125.

ezcvsc Determines the size of the largest vector arrow relative to the frame size for the `plotv` command. 47.3See “Attribute Table” on page 31, attribute `vsc`. Default: See defvsc, below.

ezclabel This string can be set to "on", "off", or "alpha" to control the kind of labels to put on contours. The default is single letters ("alpha"). A value of "on" labels each contour with its value; a value of "off" results in no labels at all.

ezcclf Format to use to form contour labels when `ezclabel="on"`. Must be a legal Fortran format, including surrounding parentheses, 16 characters maximum, no error checking. Default: `"(1Pe16.2)"`.

ezcconkey Control the display of contour level annotation. Set to "on" to show the annotation; set to "off" for no annotation. Default: on.

ezcconord Control the order of contour level annotation. Set to `"incr"` to have the values increase as one reads down the list, or to `"decr"` to display the values in decreasing order. Default: `incr`.

ezccksfill Control the contour level annotation color fill. Use "solid" for color fill or "hollow" for hollow fill. Default: hollow.

ezcfmkey Control the display of fillmesh color keys. Set to "on" to show the annotation; set to "off" for no annotation. Default: on.

ezcfmfill Control the fillmesh level annotation color fill. Use "solid" for color fill or "hollow" for hollow fill. Default: solid.

ezcarsz Multiplier for arrow size in ray plots. Default: 1.0.

ezcarsp Multiplier for spacing between arrows in ray plots. Default: 1.0.

ezcraylab Control labels in ray plots. Set to "on" to plot ray labels, set to "off" for no labels. Default: off.

ezcthickray Control the fat ray option in ray plots. Set to "on" to use ray thickness to show relative strength. Default: on.

ezcpwkey Control the display of ray power color keys. Set to "on" to show the annotation; set to "off" for no annotation. Default: on.

ezcpwfill Control the ray power level annotation color fill. Use "solid" for color fill or "hollow" for hollow fill. Default: solid.

ezciwrk Amount of integer workspace for Conpack contour routines. Default: 1000.

ezcrwrk Amount of real*4 workspace for Conpack contour routines and ARSCAM routines. Default: 5000.

ezcamap Amount of integer workspace for ARINAM routines. (Used for filled contour plots and fillmesh plots.) Default: 100000.

57.1.2 Device Control Variables

The EZD group Device_Control also contains variables which the user may find useful for fine control of the detailed behavior of his/her graphics devices.

ezccgmc Number of frames to put in a single CGM file. Default: 240.

ezcpsc Number of frames to put in a single PS file. Default: 240.

ezcdisp Xwindow DISPLAY specification. Overrides the DISPLAY environment variable. Default: a blank string.

`ezcwinwd` Xwindow width, in inches (real). Default: 7.

`ezcwinht` Xwindow height, in inches (real). Default: 7.

ezcwinlb Xwindow label specification: a string value to label the window. Default: a blank string

57.1.3 Ezcurve Default Variables

You can change some default settings by assigning new values to the following variables in group EzcurveDefaults. The types and default values are shown for each, in alphabetical order. The notations `_Size4` and `real4` mean a real value in single precision. Note: If reset, these defaults take effect after the next `nf` command, provided `ezcreset=true`. (They will have no effect if `ezcreset=false`.)

defarrow integer /NO/ - put arrows on curves? (YES=1, NO=0)

`defbnd integer /NO/ -` If `YES`, only draw boundaries of regions, not interiors. (`YES`=1, `NO`=0)

`defbot character*LSIZE /" "/ -` title for bottom. (`LSIZE`=120)

`defcolor character*16 /"fgcolor"/ -` normal color

`defcscale character*8 /"lin"/ -` default color index scale. Possible values: `"lin"`, `"log"`, or `"normal"`.

`defgridx character*8 /"off"/ -` grid lines in x direction.

`defgridy character*8 /"off"/ -` grid lines in y direction.

`defksty character*8 /"solid"/ -` style for k-lines.

`deflabel integer /YES/ -` show labels on curves? (`YES`=1, `NO`=0)

`defleft character*LSIZE /" "/ -` title for left. (`LSIZE`=120)

deflegnd integer /YES/ - show the legend? (YES=1, NO=0)

deflev integer /8/ - Minimum number of contour levels to choose (NCAR may choose ¡= twice this); negative means use logarithmic contours.

deflsty character*8 /"solid"/ - style for l-lines.

defmark character*8 /" "/ - mark – blank for curves.

defmarks real4 /1.0_Size4/ - scale size for marks.

defpoint integer /NO/ - are contour z values point-centered?

defright character*LSIZE /" "/ - title for right. (LSIZE=120)

defrsq real4 /0._Size4/ - r-squared parameter for multiquadric algorithm. (Used by random contour plots to interpolate to a grid.) Calculated for you if = 0.

defscale character*8 /"linlin"/ - default scale for axes. Possible values: "linlin", "linlog", "loglin", or "equal".

defstyle character*8 /"solid"/ - line style.

defthick real4 /1.0_Size4/ - thickness of lines.

deftop character*LSIZE /" "/ - title for top. (LSIZE=120)

defvsc real4 /0.05_Size4/ - default value for vsc, size of largest vector relative to the frame size.

ezcx Varname /"zt"/ - default name for z.

ezcy Varname /"rt"/ - default name for r.

ezcxv Varname /"vt"/ - default name for v.

ezcyv Varname /"ut"/ - default name for u.

ezcireg Varname /"ireg"/ - default name for ireg.

57.2 Parameter Access Routines

There are routines for the EZN user to *query* or *set* parameters in EZN and some NCAR packages. These features are for advanced Basis users to futher control their graphics needs.

57.2.1 Query EZN Parameters

There are three routines to query the EZN parameters:

```
ezcgeti("parameter-name",\&ival)
ezcgetr("parameter-name",\&rval)
ezcgetc("parameter-name",\&cval)
```

These routines retrieve the current EZN parameter value even if the parameter is not "dump"ed in the variable specification file (i.e., *not visible*).

The user calls one of these routines based on type of the parameter declared in the " .v file", and supplies the parameter name for the query in double quotes. The current value of the parameter will be copied into the integer, the real, or the character string variable specified in the function call.

57.2.2 Set EZN Parameters

There are three routines to set the EZN parameters:

```
ezcseti("parameter-name",ival)
ezcsetr("parameter-name",rval)
ezcsetc("parameter-name",cval)
```

Similar to the query routines, the user may call these routines to set parameters to the values provided in the calling arguments.

57.2.3 Query and Set NCAR Parameters

Similar to the discussion above, user may use cpgeti, cpgetr, cpgetc to query the parameters in the NCAR Conpack package and use cpseti, cpsetr, cpsetc to set its parameters. 48.2.2See "Contour Control Parameters" on page 40 for more information.

aggeti, aggetr, aggetc, agseti, agsetr, and agsetc are the routines for dealing with parameters in the NCAR Autograph package. 53.1See "Changing Autograph Parameters" on page 85 for more information.

vvgeti, vvgetr, vvgetc, vvseti, vvsetr, and vvsetc are the routines for dealing with parameters in the NCAR Vectors package. See 49.4"Customizing Vector Plots" on page 65 for more information.

Refer to the NCAR manuals (available on the web at URL http://ngwww.ucar.edu/ngdoc/ng/nggenrl/lludoc.html) for complete details.

Part IV

The EZD Interface

CHAPTER

FIFTYEIGHT

Introduction to EZD

58.1 Functionalities of EZD

The **EZD** package is a set of Fortran utilities for controlling graphical devices in programs that use the **National Center of Atmospheric Research(NCAR)** Graphics Library. Graphic devices supported by the EZD depend on the underlying **Graphics Kernel System(GKS)**. A computer system with **Advanced Technology Center(ATC) GKS**, the devices supported are **Computer Graphic Metafile(CGM)** files, **PostScript** files, **Xwindows**, and **Tektronix Graphics terminals**. For the computer system that has only the **NCAR GKS**, the **NCAR CGM** file is supported and additional **Xwindows** is provided if NCAR3.2 or later version is used. (For CGM and PS we use "file" or "device" interchangeably.) The EZD package also has subroutines to properly start and to end the plots, to make a frame advance, to do graphics in quadrant mode, to set up color tables, to write the log files, and to record the error log. The EZD package has stub subroutines *ezchook*, *ezcdispl* which can be replaced by a customized function routines to perform actions when a frame is advanced. A set of parameters are used in EZD to perform some specific controls such as the maximum number of frames in a CGM file, Xwindow display location, problem name of a run etc. A user may inquire the current settings of control parameters and set their values. For **UNICOS** users, EZD also provides the setup utilities to specify *Boxid*, *Security Level*, and *Give/Keep*. These setups will be used to generate proper *lpr* commands to handle user's output files.

58.2 Incorporating EZD in your program

The EZD package was designed that a client program does not need to have Basis to be loaded to generate the executables. This reduces the size of the program and broadens its usability. But because the lack of Basis support, it is necessary for the client program to call some initialization programs when EZD is invoked. The client program also needs to call functional routines to properly close the files when the client program ready to stop the execution. It needs to provide some additional functions to rendering the graphics and other related tasks. The client programs of the EZD should load the library *libezd.a* during the linking process. It is located in `/usr/local/basis/bin/lib`. You will also need to load with the appropriate NCAR and ATC libraries for your site. Refer to Section 1.1 "Environment Variables" in Chapter 1 for the list

of environment variables needed.

Furthermore, additional libraries and include files need to be specified on the command line for proper compilation of code calling EZD routines.

In the following descriptions, `[flags]` should be replaced with flags for the compiler that the user desires, `<atc device libraries>` with the desired ATC device libraries, and `<file>` with the file or files of appropriate type.

On a SUN4, this requires the following format during partial compilation *with* ATC GKS:

```
f77 -c [flags] -Bdynamic -I/usr/local/gks4101 <file.f>
```

and during linking:

```
f77 [flags] -Bdynamic -L/usr/local/gks4101            \
<files.o> /usr/local/basis/bin/lib/libezd.a          \
-lncarg -lgksflb -lgkswiss -lgksgksm                 \
<atc device libraries\> -lgksmsc -lncarv -lncarg_loc  \
-lX11
```

On a Sun4, this requires the following format during partial compilation *without* ATC GKS:

```
f77 -c [flags] -Bdynamic <file.f>
```

and during linking:

```
f77 [flags] -Bdynamic <files.o>               \
/usr/local/basis/bin/lib/libezd.a -lncarg     \
-lgksflb -lgkswiss -lgksgksm -lgksmsc -lncarv \
-lncarg_loc -lX11
```

On an HP700, this requires the following format during partial compilation *with* ATC GKS:

```
f77 -c  [flags] -I/usr/local/gks4101 <file.f>
```

and during linking:

```
f77 [flags] -Wl,-L/usr/local/gks4101 <files.o>       \
/usr/local/basis/bin/libezd.a                       \
-lncarg -lgksflb -lgkswiss -lgksgksm                \
<atc device libraries\> -lgksmsc -lncarv -lncarg_loc \
-lX11 -lm -lBSD
```

On an HP700, this requires the following format during partial compilation *without* ATC GKS:

```
f77 -c  [flags] <file.f>
```

and during linking:

```
f77 [flags] <files.o> /usr/local/basis/bin/libezd.a  \
-lncarg -lgksflb -lgkswiss -lgksgksm -lgksmsc        \
-lncarv -lncarg_loc -lm -lBSD
```

The UNICOS versions of these commands are similar. Instead of `/usr/local/gks330` we use `/usr/local/lib/ATC_GRAFPAK-GKS/gks330`.

58.3 Initialize EZD

Before invoking the functions of EZD, the client program should call `ezdinit` to initialize the EZD. It sets the parameters to its proper values. The functions of EZD depend on these values to behave accordingly.

58.4 Setting Devices

In a computer system with only **NCAR GKS** installed, a user can open CGM files to store graphic output. Addtional device support for the Xwindows when NCAR3.2 or later version is used. The CGM files created by the NCAR GKS are not standard CGM files. The NCAR CGM files can be used as input to the NCAR graphic utilities such as *ncgm2cgm*, *idt*, *ctrans* etc. For example, the utility *ncgm2cgm* translates a NCAR CGM file into a standard CGM file. *idt* lets you view the NCAR CGM file interactively. Please refer to the NCAR manuals for details about its graphic utilities. If **ATC GKS** is installed in the computer system, a user can open multiple devices and direct the graphics output to different devices. One application of these capabilities is, for example, a user can open several Xwindows at the same or different workstations, and display frames in different windows for comparison.

The graphic devices have several states: *opened*, *closed*, *active* and *inactive*. Before a device can be used, it has to be opened, then activated. Only the active devices will receive graphic outputs. Before closing a device, the device needs to be deactivated.

The subroutine *ezcdodev* is the top layer of user interface to control the devices. The calling sequence is:

```
call ezcdodev("device-arg","action-arg","modifier_arg")
```

The argument *device-arg* specifies the device that client program intends to control. The possible values are `cgm`, `ps`, `win`, `tv`, or `tek`. Here `win` and `tv` are synonymous. The argument *action-arg* indicates the actions that the user wants to perform on the device specified. The actions can be `on`, `off`, `close`,`send` and `colormap`. The argument *modifier-arg* is used to specify additional properties of the device. The values are `color`, `mono`, *window name, and colormap name*.

The underlying subroutines for the device controls are the subroutines *ezccgm*, *ezcps*, *ezcwin*, and *ezctek* control the **CGM**, **PostScript** files, the **Xwindow** and the **Tektronix Graphic Terminal** respectively. The action `on` *opens* a device if the device has not been opened and then *activates* the device. If the device already opened, the command `on` *activates* it. It has no effect on the device if it is currently active. The action `off` *deactivates* an open device (but the linkage to the device for controlling still exists). The action `close` *deactivates* and then *closes* the device. Issuing the commands `close` or `off` to a non-existing device causes an error. The action `send` *sends the current frame* to the specified device. If the target device is a CGM or a PS file, the send action turns `on` the device (even the device has not been opened before), `send` s a frame, and then turns the device `off`. If the target device is a Xwindow or a Tektronix graphic terminal, then the current frame is *resent* to the device provided the target device is *active*. The action `colormap` sets the named colormap to the device. If the device does not exist at the time when colormap action was invoked, the EZD creates the device and then set its colormap.

Due to the constraints of Xwindows driver, the colormap setting for Xwindows works differently than other devices. If ATC GKS is the underlying gks package, only the first window can be set to the desired colormap and then all sebsequent windows will inherit the colormap set by it. if NCAR GKS is used, no colormap setting is implemented because the possible program crash induced by setting the Xwindow colormap.

The *modifier-arg* is used to specify additional properties of the device. A user can use the modifier to override the default setting for the device. The command modifier `color`, `mono` overrides the default setting for CGM or PS files. CGM files default to “color” and PS files are default to “mono”. One caution with the use of color PostScript file: if it is printed at a black and white printer, the color attribute of the graphics will be plotted in different line styles; as dotted or dashed lines which may change the looks of the graphics without user’s intention. The *modifier-arg* associated with `colormap` action is used to specify the name of the colormap. There are 18 colormaps to choose from. The first 16 colormaps are named as ”`idl1`”, ”`idl2`”,...,”`idl16`”. Those colormaps defintions are borrowed from **IDL** with its **RGB**(Red,Green,Blue) settings. The 17th colormap is named as ”`mycolormap`” for user defined colormap. A user specifies `ezcred`, `ezcgreen`, and `ezcblue` arrays of RGB values to be used in the colormap. The default colormap which varies the color spectrum from blue to green to red. Any colormap name which does not match above mentioned seventeen colormap names will result to use the default colormap. An example to setup a colormap for the CGM file, call `ezcdodev("cgm","colomap","idl1")`. Another usage of the command modifier is to name the Xwindow when it is opened. The name of a window is used to identify it in future actions. As an example, `call ezcdodev("win","on","FirstWindow")` opens an Xwindow in your default workstation and names the window `FirstWindow`. If multiple windows were used, *ezcwin* activates only *one* window. The latest window with action `on` is the *active* window. The

activated window receives graphic output. (If multiple *displays* were used, you may have one active window in each *display*.) Because of this arrangement, a user can direct the graphic output to different windows in order to view and to compare the graphic results interactively.

When a CGM or a PS device is opened, the file name will be determined by the file root name "`fnroot`" parameter with extension `.cgm` or `.ps` correspondingly. The default file root name is `"problem"`. The subroutine *ezccgm* and *ezcps* check (in the current directory) the existence of specified files (either by the user or by the program defaults). If the file already exists, the subroutine tries to append a sequential three digit number to the root name to make a new file name. This avoids clobbering the existing files e.g. problem.001.cgm, problem.002.cgm etc. EZD will create a CGM log file or a PS log file for each CGM/PS file created to record the frame count and all graphic commands in each frame. The naming scheme for the log files is the same as for CGM or PS files but with extension `.cgmlog` or `.pslog`.

58.5 Starting and Ending the plots

The subroutines *pltstart* and *pltend* are provided to properly open and close graphic device(s) for receiving plot commands. They are argumentless routines.

```
call pltstart
call pltend
```

The *pltstart* checks the existence of active devices. It does nothing if an active device already exists. It opens a CGM file automatically if no active devices were found. The *pltend* closes all devices and log files. *pltstart* should be called before any plotting commands and the client program should call *pltend* to terminate graphics before program ends. Explicitly close device by calling *ezcdodev* or implicitly by calling *pltend* is important to leave the device in *proper state*. For example, if CGM file was not properly closed, it will cause command `lpr` to fail on the UNICOS systems.

58.6 Quadrant mode

EZD has utilities to control the plots in *quadrant mode*. Most quadrant routines have companion routines for inquiring the current settings.

The subroutine `ezcdquad(xmin,xmax,ymin,ymax)` defines the quadrant reference box. *xmin*, *xmax*, *ymin*, *ymax* are reals in [0.,1.] as the bounding values of a rectangular region in the frame which has the values (0.,1.,0.,1.). The default reference box of quadrant is the **whole** frame.

The routine `ezcidquad(v1,v2,v3,v4)` returns the current boundary of the quadrant reference box.

The quadrants are defined in a customary way by **bisecting** both the length and the width of the reference box. The quadrant 1 is the upper-left quarter. The quadrant 2 is the upper-right quarter. The quadrant 3 is the lower-left quarter and the quadrant 4 is the lower-right quarter.

The `subroutine ezcquad(n)` where n is one of the following integers: 1, 2, 3, 4, 12, 13, 24, 34, and 1234 defines a combination of quadrants represented by each digit. For example, `call ezcquad(12)` defines the region combined with quadrant 1 and 2 to draw the graphic output. `call ezcquad(1234)` is the same as the original whole reference box.

Another example, after called `ezddquad(0.0,0.5,0.5,1.0)`, then the call to `ezcquad(1)` would set the first quadrant of now just defined reference box, i.e. the most upper left one-sixteenth of the original frame for plotting graphics.

`ezcsquad(xmin,xmax,ymin,ymax)` allows a user to set an arbitrary rectangular portion of the **whole** frame bounded by the specified arguments to plot the graphics. The companion `subroutine ezciquad(v1,v2,v3,v4)` can be used to inquire the current quadrant boundary. The call to `subroutine ezcrquad` will restore the reference box defined by *ezcdquad* as the plot region.

58.7 Frame Advance

The frame advance logic is complicated because of the need to lag the actual frame advance for interactive window use and the need to handle quadrant/non- quadrant graphics. The action routine *ezcfradv* does the frame advance if the flag *ezcxn* has been set to 1 (i.e. YES). The routine *ezcnq* displays the current picture by calling *ezcshowf* which calls *ezcdispl* to flush out the graphic contents and calls *ezcfradv* to advance the frame if needed. The *ezcdispl* to flush out the graphic contents is a dummy routine in the EZD and need be supplied by the client program. For each new frame, the client program should call *ezcnf* which sets the flag *ezcxn* to 1 and calls *ezcnq*. After the frame advanced, *ezcnf* resets the flag *ezcxn* to 0 and gets it ready for the next event.

58.8 Error Logging

Error logging facility in the EZD library is performed through the *ezcerror* subroutine. `ezcerror("msg",level)` accepts a quoted string and an integer error severity level as its arguments. The subroutine writes the string to the **standard error file**. **Three levels** of error severity are defined as follows: level one is a commentary, level two is a minor abnormality, the program continues to execute, level three is a fatal error, and it calls the user defined error handler *ezcdie* (the client program may wish to provide this) to manage alternative actions response to the sever errors.

When a user calls this routine to record the error message, he/she also needs to determine the severity of the error and assign the severity level accordingly.

58.9 Color Table

`ezcoltb(indlo,indhi,red,green,blue)` defines color tables

for all **active** devices. *indlo* is the integer for *lower bound* of color indices, and *indhi* is the *upper bound* of color indices, and *red, green, blue* are arrays of fractions of full intensity of red, green, and blue color range in [0., 1.]. With a user defined color table, applications may use special colors to convey some physics quantities such as color cells.

58.10 Set a Predefined Colormap/Color Table

`ezcdodev("device-type","colormap","colormap-name")`

The routine `ezcdodev` can be used to setup a special colormap for a device. If the device does not exist (i.e. has not been `opened`) at the time of the subroutine call, then **EZD** will *open* the device, *activiate* the device and then *setup the requested colormap*. If the device already exits, then colormap is *reset* and then *returned* to its *original state* (e.g. active, inactive) just before the `ezcdodev` was invoked. There are sixteen predefined colormaps named "`idl1`","`idl2`",...,"`idl16`" which are borrowed form the **IDL**'s colormap definitions. The "`idl1`" colormap is the *greyscale* colormap, so the user may use "`greyscale`" as the colormap name. The "`idl2`" has alias "`bluescale`", but some idl colormaps have no proper aliases. The seventeenth colormap named "`mycolormap`", is defined by the user's specifications to the arraysof `ezcred`, `ezcgreen`, and `ezcblue` for its RGB values. Any other name used for colormap will result to use the default colormap, which is a colormap varies the color spectrum from blue to green to red.

Some exceptions when Xwindow and postscript files are involved. The color postscript file and mono postscript file can not switch from one to another. Only color postscript file can change its colormap. The Xwindow device driver from ATC allows the *first* Xwindow to set its colormap first time when it is brought up then the subsequent Xwindows will share the same colormap. For a single Xwindow to change its colormap, the EZD actually closes the window then opens it again with the new colormap. The NCAR Xwindow driver currently will cause the application program to crash if change colormap is attempted. So EZD will not allow any colormap setting for Xwindows in the applications without ATC GKS.

58.11 Box, Security Level, and Give/Keep

For the programs run on the **UNICOS** systems at **Livermore Computer Center**, it is required to specify a *Boxid*, *Security Level*, and *Give/Keep* for the output. The subroutines *ezcdobox*, *ezcdolev*, and *ezcdogk* are provided to set them up. The calling sequence for these routines is

```
call ezcdoxxx("string")  where xxx=box, lev, or gk
```

The argument is a string or a variable with string value. *ezcdobox* accepts a three character string as the argument. The first character is an alphabet among a-z and A-Z, then followed by two alphanumeric characters. *ezcdolev* accepts the argument with the possible values of `uncl`, `UNCL`, `pard`, `PARD`, `crd`, `CRD`, `srd`, `SRD`, or numerical characters `1`, `2`, `4`, and `5` which corresponding to *unclassified*, *pard*, *crd* and *srd*. The argument for *ezcdogk* takes either `give` or `keep`. It defaults to `keep` if not explicitly specified.

When the CGM file closes, the above setups will be used to send a proper `lpr` command to the **UNICOS** operation system to produce fiche. For example, say the Boxid=u51, Severity Level=pard and Give/Keep=keep, the EZD will issue the following command string:

```
lpr -P105 -Bu51 -Spard problem005.cgm
```

when the CGM file problem005.cgm is closed. A default jobname same as the CGM file name is also provided to the `lpr` command to print it on the fiche.

58.12 Stub Routine - ezchook

A stub routine named *ezchook* is called by the *ezcfradv* subroutine with syntax

```
call ezchook("string1","string2").
```

The *ezcnf* calls *ezcfradv* to advance a frame. A user can substitute this stub routine *ezchook* with his/her own subroutine to perform customized tasks when each frame advances.

58.13 Access to Parameters - ezcseti, ezcsetr, ezcsetc, ezcgeti, ezcgetr, ezcgetc

A set of parameters in the EZD package provides special controls to the graphic devices such as maximum number of frames in a CGM file, the Xwindow display workstation other than the default environment variable setup, the root name of the problem etc. Six routines *ezcseti*, *ezcsetr*, and *ezcsetc* are used to set integer, real, character parameters correspondingly. The subroutine *ezcgeti*, *ezcgetr* , and *ezcgetc* are used to inquire and to retreive the current value of a parameter. To access an *array parameter*, the user needs to specify the *index* of the array element by calling `call ezcseti("ezcpidx",ivalue)` before the call for “set” or “get” the parameter.

The arguments to these subroutines all have the same format,

```
call ezcsetx("parameter-name",parameter-value), where x=i,r,c
call ezcgetx("parameter-name",parameter-variable), where x=i,r,c
```

the first argument contains the name of the parameter variable enclosed by double quotes, the second argument is the value you want to set for the parameter or the variable to receive the value of the parameter.

If the user gives a non-existing parameter to the subroutine (including misspelled name), the subroutine produces an error message and then exit. The client program needs to define an error handler to respond this situation.

Following is a list of control parameters, their function and default value:

name	**brief Descriptions**	**default value**
ezccgmc	maximum number of frames in a CGM file	242
ezcpsc	maximum number of frames in a PS file	242
ezcdisp	string to specify Xwindow display	yourhost:0.0
ezcwinsz	string to specify size of Xwindow	-dx -dy -u
ezcwinlb	string to name an Xwindow	blank string
numcol	number of color indices in a color table	192
fnroot	root name used for the CGM/PS files and log files	”problem”
debcolr	debugging flag for color problems	0

-

CHAPTER

FIFTYNINE

List of Subroutines

This chapter contains a list of subroutines and their arguments. The subroutines are sorted by name. A brief description of each routine is also attached.

59.1 ezcapsfx

Calling Sequence

```
subroutine ezcapsfx(namer,ftype,fnsfx,fname,succ)
```

Description

Append suffix to a given file root name. This routine is called by `ezcwin` and `ezcps` to open a file with unique name.

Arguments

`namer` character*(80), the file root name

`ftype` character*(16), the file type, e.g. cgm, cgmlog, ps, pslog

`fnsfx` integer, the file name suffix as integer

`fname` character*(80), returned unique file name

`succ` logical, success flag of the subroutine process

Procedure

Append the fnsfx to namer if fnsfx is less than 999, otherwise change fnsfx to 1 and extend the file root name with ending "." then append the new fnsfx. If the extension of the root name failed, the return flag succ is set to false.

59.2 ezccgm

Calling Sequence

```
subroutine ezccgm(iflag,istring)
```

Description

Control the CGM devices. This is an underlying subroutine called by `ezcdodev`

Arguments

`iflag` character*(*),action-command-string, the possible values are `on`, `off`, `send` or `close`.

`istring` character*(*), command-modifier-string, the possible values are `color`, `mono`.

Procedure

For the action command “on”:

Open and activate a CGM device if no existing CGM device. Assign a proper file name for the CGM file, and open a CGM log file if it does not exist. Activate a CGM device if it has been deactivated. No action if an active CGM file exists.

For the action command “off”:

Deactivate the current active CGM device. No action if no active CGM device.

For the action command “send”:

Turn the CGM device “on”, “send” a frame, then turn the CGM device “off” (“send” implies “on” so it may open (i.e. create) a CGM file)

For the action command “close”:

Deactivate and then close the CGM file. If the client program runs on UNICOS at LC, proper `lpr` command will be send to the operating system to generate fiche from the close CGM file.

The command-modifier “color”, “mono” specifies the CGM file color.

59.3 ezccidx

Calling Sequence

```
subroutine ezccidx(iws,iwstype)
```

Description

Initialize the color indices table, define foreground and background colors, define a set of named colors with special indices.

Arguments

iws integer, the workstation id associated to this special color table

iwstype integer, the type of this workstation

Procedure

Define the color index 0 and color index 1 with RGB values. The color index 0 is the background color and the color index 1 is the foreground color. The subroutine sets “black” as the background and “white” as the foreground if the variable `bakcol` has value 0 and reverse the setting if `bakcol` has value 1. Other color indices and associated RGB values are defined in the process. This setting is closely coupled to `ezcctoi` subroutine call. The color index returned by `ezcctoi` by the giving color name has the RGB value defined in this subroutine.

59.4 ezcclear

Calling Sequence

```
subroutine ezcclear
```

Description

A dummy routine called by ezcnf. The original usage is to clear the attribute settings. Since this routine is called by every frame advance, it is user replaceable to do some customized tasks.

Arguments

none

Procedure

none

59.5 ezccoltb

Calling Sequence

```
subroutine ezccoltb(indlo,indhi,red,green,blue)
```

Description

Set a set of color indices with the given RGB values.

Arguments

indlo integer, lower bound of the color indices

indhi integer, higher bound of the color indices

red, green, blue – real(indhi-indlo+1), arrays of reals in [0., 1.] of fractions of full intensity of the red, green, blue colors.

Procedure

Set the color table with indices vary from indlo to indhi. Each index associates a color defined by the corresponding indexed array element of red, green, and blue.

59.6 ezcctoi

Calling Sequence

```
subroutine ezcctoi(colorname)
```

Description

Based on the given colorname returns the corresponding color index in the color table.

Arguments

`colorname` character*(32), colorname string

Procedure

Search the colorname array for the given colorname. If a name matched, returns the index in the colorname array. If no name is matched, returns the index 1.

59.7 ezcdodev

Calling Sequence

```
subroutine ezcdodev(devtype,arg1,arg2)
```

Description

A top layer user interface routine to control the graphics devices. This subroutine redirects device control commands to the specific device control routine such as `ezcwin`, `ezcps`, `ezccgm` etc.

Arguments

`devtype` character*(*), the device type that will receive the control commands, the possible values are `cgm`, `ps`, `win`, `tv`, and `tek`.

arg1 character*(*), the action-command-string, the valid commands are `on`, `off`, `send` or `close`

arg2 character*(*), the command-modifier-string, the possible values are `mono`, `color` or a *window name*. When the device is a *cgm* or a *ps*, user can specify color options for the device. The *cgm* is default to "color" and the *ps* is default to "mono". When the device is a *win*, this command-modifier-string can be used to set the window name.

Procedure

Based on the given device type, this subroutine redirects the command and its modifier to call the subroutine that handles this special device. For example,

```
call ezcdodev("cgm","on","color")
```

will call the underlying subroutine `ezccgm("on","color")` to carry out its command.

59.8 ezcsquad

Calling Sequence

```
subroutine ezcsquad(v1,v2,v3,v4)
```

Description

Set a rectangular portion of the frame to plot the graphics, frame size will not be changed. (vs. *ezcframe* which resets the frame boundary)

Arguments

v1, v2, v3, v4 real(Size4), the xmin, xmax, ymin, ymax values in [0., 1.] of the rectangular region of the frame

Procedure

This routine calls `ezcnq` to handle frame advance and flush out graphic contents if necessary. After the call to `ezcnq`, it sets the portion of frame as defined by `v1`, `v2`, `v3`, `v4` to output the graphics.

59.9 ezciquad

Calling Sequence

```
subroutine ezciquad(v1,v2,v3,v4)
```

Description

Inquire the current quadrant boundaries.

Arguments

v1, v2, v3, v4 real(Size4), the xmin, xmax, ymin, ymax values in [0., 1.] of the rectangular region of the frame set by the last call to ezcsquad.

Procedure

Retrieve the boundary values from the common block.

59.10 ezcquad

Calling Sequence

```
subroutine ezcquad(iquad)
```

Description

Based on the reference box for quadrants, ezcquad(iquad) will set plotting quadrant to the combinations of customary quadrants 1, 2, 3, 4 start from the upper-left corner, upper-right corner, lower-left corner and lower-right corner.

Arguments

iquad integer, one of the following values 1, 2, 3, 4, 12, 13, 24, 34, and 1234

Procedure

This is a short cut to define plotting quadrant to one of the customary quadrant or a combination of customary quadrants by calling `ezcsquad` with proper xmin, xmax, ymin and ymax.

59.11 ezcdquad

Calling Sequence

```
entry ezcdquad(v1,v2,v3,v4)
```

Description

Change the default reference box for quadrants to (v1, v2, v3, v4)

Arguments

v1, v2, v3, v4 real(Size4), xmin, xmax, ymin, ymax values in [0., 1.] with respect to the full frame

Procedure

Set boundary limits to v1, v2, v3, v4 and call NCAR “set routine”

59.12 ezcidquad

Calling Sequence

```
entry ezcidquad(v1,v2,v3,v4)
```

Description

Inquire the default reference box for quadrants.

Arguments

v1, v2, v3, v4– real(Size4), xmin, xmax, ymin, ymax values in [0., 1. respect to the full frame]

Procedure

Retrieve the values stored in the common block for reference box for quadrants.

59.13 ezcrquad

Calling Sequence

```
entry ezcrquad
```

Description

Restore quadrant to the default reference box for quadrants. Refer to ezcdquad

Arguments

none

Procedure

Reset the view port to the last defined reference box for quadrants and set up linear ndc transformation into viewport by calling NCAR “set routine”

59.14 ezcdie

Calling Sequence

```
subroutine ezcdie
```

Description

This is a routine called by EZD routines when abnormal conditions are encountered. This routine raises signal SIGUSR1. The client program should provide alternative actions when the signal is received. Hence this is the interface for the client program exception handler.

Arguments

none

Procedure

Raise the signal SIGUSR1 by calling C routine `raise`.

59.15 ezcdispl

Calling Sequence

```
subroutine ezcdispl
```

Description

This is a stub routine that the client program should replace with its real procedure to flush out the graphic contents. The routine is called by `ezcfradv` which in turn is call by `ezcnf`.

Arguments

none

Procedure

A dummy routine as a place holder. It is called for each frame advance. The user may substitute it with special task routine such as display the graphic contents, write a log entry etc.

59.16 ezcdobox

Calling Sequence

```
subroutine ezcdobox(boxid)
```

Description

Defines the Boxid for the LC UNICOS user to receive output.

Arguments

`boxid` character*(3), three character string to identify the boxid, first character is an alphabet in a-z,A-Z, and the last two characters are two alphanumeric characters.

Procedure

The routine checks the legality of the input string and write it to a common block holding this value. ezccgm will grab this string value from the common block to issue proper `lpr` command to the operating system.

59.17 ezcdogk

Calling Sequence

```
subroutine ezcdogk(gkstring)
```

Description

Defines the GIVE/KEEP flag for the LC UNICOS user to allow disposal of the CGM files after fiche output is generated.

Arguments

`gkstring` character*(*), character string to set the GIVE/KEEP flag to “give” “givekeep” or “keep”

Procedure

The routine verifies the input string and write it to a common block holding this value. ezccgm will grab this string value from the common block to issue proper lpr command to the operating system. The default value is “keep”.

59.18 ezcdolev

Calling Sequence

```
subroutine ezcdolev(lvstring)
```

Description

Defines the Security Level for the output for LC UNICOS users.

Arguments

`lvstring` character*(6), character string to identify the security level, they are “uncl”/ “UNCL”, “pard”/“PARD”, “crd”/“CRD”, “srd”/“SRD” or “1”, “2”, “4”, “5” correspondingly.

Procedure

The routine verifies the input string associates with its boxid (for classified output to a proper box) and write it to a common block holding this value. ezccgm will grab this string value from the common block to issue proper `lpr` command to the operating system.

59.19 ezcerror

Calling Sequence

```
subroutine ezcerror(msg,sevlev)
```

Description

The routine records the error message to STDERR and calls ezcdie to send signal if fatal error occurs.

Arguments

`msg` character*(120), error message

`sevlev` integer, error severity level, level 1= comment, level 2= minor abnormality, level 3= fatal error

Procedure

Copy the error message to the STDERR file and call ezcdie if sevlev = 3. You may work to replace ezcdie or supply a signal handler for the signal raised by it. See ezcdie.

59.20 ezcfradv

Calling Sequence

```
subroutine ezcfradv(note)
```

Description

Perform frame advance if the flag `ezcxn`has been set to 1.

Arguments

`note` character*(*), string input to the stub routine `ezchook`.

Procedure

The routine checks the value of `ezcxn` then either do frame advance or just return to the calling program. Before exitting, `ezcfradv` calls `ezchook` with “note” as the second argument. The user can define `ezchook` to perform customized tasks.

59.21 ezcgetcl

Calling Sequence

```
subroutine ezcgetcl(w)
```

Description

This routine returns the index of last non-blank characters in the given string w.

Arguments

`w` character*(*), string needs to determine the index of last non-blank character.

Procedure

Finds the rightmost occurrence of non-blank character.

59.22 ezchook

Calling Sequence

```
subroutine ezchook(msg1,msg2)
```

Description

This is a stub routine to be replaced by a true function routine supplied by the client program when each frame advances.

Arguments

`msg1`, `msg2` character*(*), strings used to pass the arguments to the true function routine.

Procedure

A stub routine.

59.23 ezcnf

Calling Sequence

```
subroutine ezcnf()
```

Description

Makes a frame advance.

Arguments

none

Procedure

The `ezcnf` routine calls `ezcnq` to do frame advance if the external flag `ezcxn` has the value 1. It also clears the flag `ezcxn` and restore the full *reference box* for quadrants as the plotting area.

59.24 ezcnq

Calling Sequence

```
subroutine ezcnq()
```

Description

Does actual call to do frame advance if not in the *quadrant mode*. It also displays the graphic contents by calling `ezcshowf`. (`ezcshowf` in turn calls `ezcdispl` to do the display business. In EZD, the `ezcdispl` routine is a dummy routine as a place holder. The client program of EZD has to provide a true function routine for displaying the graphic contents.)

Arguments

none

Procedure

call `ezcshowf` to display the graphic contents. call `ezcclear` to clear the collection. In EZD `ezcclear` is a stub routine that the client program can replace it for real task. It will stay in the *current quadrant* when in *quadrant mode*.

59.25 ezcps

Calling Sequence

```
subroutine ezcps(iflag,istring)
```

Description

Controls the PS devices. It is one of the underlying action routines for `ezcdodev`.

Arguments

`iflag` character*(*), action-command-string, the possible values are `on`, `off`, `send` or `close`.

`istring` character*(*), command-modifier-string, the possible values are `mono`, `color`.

Procedure

For the action command “on”:

Open and activate a PS device if no existing PS device. Assign a proper file name for the PS file, and open a PS log file if it does not exist. Activate a PS device if it has been deactivated. No action if an active PS file exists.

For the action command “off”:

Deactivate the current active PS device. No action if no active PS device.

For the action command “send”:

Turn the PS device “on” send a frame, then turn the PS device “off” (“send” implies “on” so it may open(i.e. create) a PS file)

For the action command “close”:

Deactivate and then close the PS file.

The command-modifier “color”, “mono” specifies the PS file color. The PS file has default “mono” for its color specification.

59.26 ezcsetbb

Calling Sequence

```
subroutine ezcsetbb()
```

Description

This subroutine sets the background color to black(default).

Arguments

none

Procedure

The routine sets the `bakcol` value to 0, then `ezccidx` based on this value to set background color to black. The client program should call this routine just before opening the device which will have the desired background color. (Currently this feature has been disabled due to the color table problems in the Xwindow driver)

59.27 ezcsetbw

Calling Sequence

```
subroutine ezcsetbw()
```

Description

This subroutine sets the background color to white.

Arguments

none

Procedure

The routine sets the `bakcol` value to 1, then `ezccidx` based on this value to set background color to white. The client program should call this routine just before opening the device which will have the desired background color. (Currently this feature has been disabled due to the color table problems in the Xwindow driver)

59.28 ezcshowf

Calling Sequence

```
subroutine ezcshowf
```

Description

Display the current picture and set a new frame if not in quadrant mode.

Arguments

none

Procedure

The routine checks for the plotting mode first. If it is in quadrant mode then no frame advance, i.e. does not clean the frame so the previous picture on the frame remains. If it is not in quadrant mode, then frame is advanced. It is then plot the graphics to the frame by calling `ezcdispl`. In the EZD, this `ezcdispl` is just a stub routine which the client program should replace it by a true action routine.

59.29 ezcshowg

Calling Sequence

```
subroutine ezcshowg
```

Description

Invoking graphic display routine `ezcshowf`. In EZD, it is a dummy routine as a space holder.

Arguments

none

Procedure

Invokes `ezcshowf` to display the graphics contents. In EZD, it is a dummy routine and should be replaced by a true display action routine or invokes `ezcshowf` to indirectly display the graphic contents. The command “send” depends on this routine to flush a frameful graphics to the designated device. (e.g. `call ezcdodev("cgm","send","color")`)

59.30 ezctek

Calling Sequence

```
subroutine ezctek(iflag,istring)
```

Description

Controls the Tektronix graphic terminal devices.

Arguments

`iflag` character*(*), action-command-string, the possible values are `on`, `off`, `send` or `close`.

`istring` character*(*), command-modifier-string, has not been used by this special device. (Keep the argument just for consistency with other devices.)

Procedure

For the action command “on”:

Open and activate a Tektronix graphic terminal device if no existing Tektronix graphic terminal device. Activate a Tektronix graphic terminal device if it has been deactivated. No action if an active Tektronix graphic terminal exists.

For the action command “off”:

Deactivate the current active Tektronix graphic terminal device. No action if no active Tektronix graphic terminal device.

For the action command “send”:

Turn the Tektronix graphic terminal device “on” send a frame, then turn the Tektronix graphic terminal device “off”

For the action command “close”:

Deactivate and then close the Tektronix graphic terminal.

The command-modifier is not used for the Tektronix graphic terminal.

59.31 ezcwin

Calling Sequence

```
subroutine ezcwin(iflag,istring)
```

Description

Controls the Xwindow devices.

Arguments

`iflag` character*(*), action-command-string, the possible values are `on`, `off`, `send`, or `close`.

`istring` character*(*), command-modifier-string, a window name string

Procedure

The command-modifier istring provides the window name as the way to identify the recipient of the action-command.

For the action command “on”:

Open and activate an Xwindow device with the given window name if no existing Xwindow device with the same name. This will deactivate other active windows in the same display.

Activate an Xwindow device if the named window was created before and has been deactivated. No action if the named Xwindow already active.

For the action command “off”:

Deactivate the named Xwindow device. No action if the named Xwindow is not active. It is an error try to “off” no existing Xwindow. (e.g. no window with the given name hence can not be “off”ed.)

For the action command “send”:

Send a frame to the active Xwindow.

For the action command “close”:

Deactivate and then close the named Xwindow. It is an error to try to “close” a non-existing Xwindow.

If there is only one window, by default it receives the action command. -

Part V

Writing Basis Programs, A Manual for Program Authors

CHAPTER

SIXTY

Basis Development Overview

As mentioned before, Basis is both a program and a developement system. Basis the language is documented in Part II, Basis Language Reference. This part deals with Basis as a development system.

The build system consists of `dsys` and `mio`. The `dsys` script is developer's interface to build Basis. Internally `dsys` uses `mio` to generate `make` files to do the actually compiling and loading.

Once Basis is installed, the utility `basiskit` can be used to create the scaffold necessary to build a simple Basis program. The next chapter deals with building more sophisticated Basis programs.

Basis has a fairly simple type system. The database keeps track of type (integer, real, logical) and size (4 or 8 bytes). The `configcompiler` and `typeheaders` scripts are used to match the native compilers types to Basis' types.

`Fcc` is used to create wrappers that allow Fortran to call C functions. The generated wrappers deal with name-mangling and call-by-reference/call-by-value differences between Fortran and C.

At Basis' core is a runtime database that contains information about variables and functions. The `mac` program reads a Variable Description File and creates the code necessary to intern information about variables and function into the database. It also creates handlers to allow functions to be called from an interpreter. In addition `mac` creates files that allow Fortran and C compilers to access the variables directly. The Basis interpreter has access to the runtime database. This allows the interpreter to access the user's variables and call compiled and builtin functions. The `gluepack` utility creates code to put packages into a single Basis executable.

CHAPTER

SIXTYONE

Installing Basis

61.1 Install Overview

The Basis source directories are organaized as

```
basis/rt: the Basis run-time package
      ...
basis/scripts: cfgman, cpu, mio  ...

basis/builder: dsys, the ''heart'' of the Basis build
basis/builder/std:   generic config files
basis/builder/local: custom config files
basis/builder/features:

basis/test: test files and repository of fiducials
```

Once a config file has been created, Basis is compiled with the sequence:

```
dsys config input
dsys build
dsys test
```

61.2 Build Details

There are four overall stages in making a Basis program:

1. For each variable descriptor file, run the `mac` program to create the connections between the variable descriptor file, the source, and the runtime database. This will also create connections to any C or C++ code that may be present.

2. Compile the resulting output, precompile and compile each MPPL source file, and compile each Fortran, C, and C++ file. For each directory, containing one or more packages, a single object file or library is created.

3. Run the `gluepack` program to create the connection between Basis and the desired packages.

4. Link the program with the Basis run-time library and any desired graphics libraries and user-specified libraries.

CHAPTER

SIXTYTWO

Dsys: Automating Building and Testing

`Dsys` is a script that provides a coherent interface between code developers and the various code management utilities. Many large scale code projects deal with a variety of tools including compilers, linkers, make, and source management utilities such as `cvs`. These tools have many options and details with which most developers would rather not have to concern themselves.

On the other hand, most developers have a high level idea of what it means to compile and link their codes, or to commit their changes. So `dsys` bridges that gap by defining a set of high level operations such as config, build, and commit each with a few simple options. The details of these high level operations are then carried out by `dsys`. These details are worked out once and define the procedures by which a code system is to be managed. The script also serves as documentation of the procedures.

62.1 Dsys Targets

The following list of `dsys` targets gives some of the high level operations to illustrate the extensive capability `dsys` makes available to the Basis developer. These are a few of the once common to many code systems. For a complete listing, see the `dsys` man page.

build The code system is compiled, usually governed by make, and any executables are linked.

commit Changes to the code are committed into a source repository.

config A code system is configured to be built on a particular platform with various options/

dist A distribution such as a tar file of the sources is made for transport to other systems

help (or -h) Give information about dsys options.

info Information about the sources or any aspect of the code system is found and printed out.

install The code system is installed for public use as opposed to private development

link Link the Basis executable.

remove Binaries files such as objects, archives, and executables are removed.

sync The sources being developed are brought up to date with the sources in a repository

test Tests for the code system are run to verif the code.

New targets are added to `dsys` constantly. `Dsys` has a help option that will list its targets and most of these targets also have a help option which describes options specific to that particular target. In practice the builder directory is added to the source tree to contain `dsys` and any scripts, configuration files, or other information needed to manage the code system. In this way all this information is together and separated from source files that may be compiled or operated on by tools controlled by `dsys`.

CHAPTER

SIXTYTHREE

MIO: Make is OK

In modern software systems, the process of compiling and linking correctly on a wide variety of platforms can be a difficult problem. When working on multiple platforms simultaneously, it is highly desirable to use just one copy of the source yet produce output for many different machines. A general solution of this problem is difficult, but we have provided a Basis-specific solution which should fit the needs of most authors of Basis programs. This utility is called `mio`.

`Mio` consists of two logical parts. First it reads a series of input files and builds up an internal database. Second it write out files necessary to build the code based on the database.

`Mio` will automatically construct platform-specific `pre-Make` files that will be used as input to the Unix utility `make` to build your code on multiple platforms. Typically `mio` executes in just a few seconds.

A manual page for `mio` is available in `BASIS_ROOT/man/man1`.

Since version 12.0 of Basis, a utility, `mio`, is usually used to automate most of the compile-load cycle. In addition to `man` pages for `mio` which come with the Basis distribution (`mio` and `mio-intro`), this manual explains the use of `mio` too. Versions of Basis prior to version 12.0 used a utility `mmm`, but this utility is no longer supported, and its use is strongly discouraged. The Basis team has tried hard to provide documentation to help make conversion to the new methodlogy as simple as possible.

63.1 Mio Overview

Using information from a configuration description file (config file) and/or a Basis `Package` file, `mio` generates other files which are used in conjunction with various system utilities to manage the compilation and linking of a Basis code. The goal is to be able to build Basis codes, including Basis itself, on multiple, different computer systems simultaneously.

`Mio` will read a configuration file which describes the details of the specific compilation of the Basis code you desire. `mio` will set up directories to hold: executable files (`bin`); library archive files (`lib`); files used by compilers and interpreters (`include`); documentation (`man`); and log files from compilations and other operations typical of a code system (`log`). It will produce files which help govern the compilation and installation of a code as well as a file called `configured`

which is a record of how the code system was last configured for a particular platform.

63.2 MIO output files

mio is capable of writing many output files. The name of the file is controlled by a variable. If the name is blank, the file is not created.

63.2.1 configured

A summary of the final configuration. Has all C¡Use¿ statemtents expaned.

63.2.2 configured.pl

A perl readable version of the config database.

63.2.3 code-m-def.d

Used by mppl source. creation controlled by `Write_m_defs`. Defines controlled by `VMDef`.

63.2.4 code-f-def.d

Used by fortran source. Creation controlled by `Write_f_defs`.

63.2.5 code-c-def.d

Used by C source. Creation controlled by `Write_c_defs`. Defines controlled by `VCDef`.

63.2.6 make-config

Used by **pck** to build the final makefile. Creation controlled by `Write_make_config`. Defines controlled by `VMake`.

63.2.7 Makefile

Global Makefile used to compile code in parallel. Creation controlled by `Write_makefile`.

63.2.8 mio.csh

Used by csh. Creation controlled by `Write_mio_csh`. Setenv controlled by `VEnv`.

63.2.9 mio.make

Used by make. Creation controlled by `Write_mio_make`. Setenv controlled by `VEnv`.

63.2.10 mio.pl

Used by perl. Creation controlled by `Write_mio_pl`. Setenv controlled by `VEnv`.

63.2.11 mio.sh

Used by bourn shell. Creation controlled by `Write_mio_sh`. Setenv controlled by `VEnv`.

63.2.12 packages

List of package names. Creation controlled by `Write_packages`.

63.2.13 Packages

Package groups for use by other codes. Creation controlled by `Write_Packages`.

63.2.14 pre-Make

In a package-level directory mio creates directory *$cpu* if it doesn't alread exist and *$cpu/pre-Make*. The generic targets defined in the pre-Make file are: remove, build, mac.

The build target compiles files, build archives, and depending on the package level configuration links any executables specified. The `mac` target runs the `mac` utility over any .v files specified. This is called out as a separate step to control dependencies and enable parallel make operations to succeed.

63.2.15 pck

The **pck** file is a trivial script that determines which platform it is being run on and then goes to the appropriate *$cpu* directory to do the requested operation. To do a clean build of the package with an mio configured code you might do:

```
pck remove
pck build
```

regardless of the platform you are on.

63.3 MIO syntax

Variables and groups are the two data structures of `mio`. Variables are simply a name and a value. Groups are collections of Variables. Groups also have a class associated with them.

Any line where the first non blank character is a octothorpe (#) is treated as a comment.

63.3.1 Variables

The syntax for defining and assigning variables in config files is fairly simple. There are three forms:

```
var  = value
var += value
var -= value
```

If *value* contains the pattern `$@` the current value of *var* is substituted at that point.

Leading blanks are removed. `var = value` results in `var` being assigned "`value`", not "` value`". Leading blanks can be assigned using the {} syntax below.

When appending, a blank and `value` are added to the current value of `var`.

```
Flags  =-g
Flags +=-o
```

results in `Flags` being assigned `-g -o`.

If the first character of `value` is a open curly brace ({), then all text up to the balanced closing curly brace, excluding newlines and comments, are assigned to `var`.

```
var = {
       value1   # comment about value1
       # other values
       value2 value3
      }
```

results in `var` being assigned "`value1 value2 value3`".

```
var << END
```

Here document form. All text upto a line starting with the string `END`, including newlines and comments, are assigned to *var*. `END` may be any user defined string.

`mio` generates some variables names that begin with an underscore.

63.3.2 Groups

Groups collect variables into a new namespace. A Group is created by `name : class`.

```
class : name {
  Flags = -flag
}
```

Group names are any sequence of letter, numbers or special symbols. `code`, `1`, `file.c` are all valid group names.

Additional references to *name* will add to the group.

```
class : name {
  Flags = -flag
}
class : name {
  Flags += -flag2
}
```

Name is optional.

```
class {
  Flags = -flag
}
```

Currently the group is assigned the name `--anon--`.

63.3.3 Functions

`Mio` also has functions to allow operations on variables, groups, and the environment. Functions are a name followed by a set of parentheses enclosing any arguments. The parentheses are required even if no arguments are specified.

clear Delete all variables in the current namespace. Does not work in the global namespace.

delete(name) Delete variable *name* from the current namespace.

error(msg) Write *msg* to the screen and exit. Arguments are expanded before printing *msg*.

expand(string, variable) *String* is string interpolated and the resulting value is put in *variable*. A `$` is used to indicate variable expansion.

```
input = Hello
expand($input world, out)
```

Results in `out` being assigned "`Hello world`".

export(variables) Take the list of *variables* and set them in the current enviroment. This is one way of passing current database values to programs executed by the **run** function. Environment variables have the form `M__`*variable*.

variables is expanded before exporting. If it is a blank delimited list, then each name will be exported.

If *variable* has a colon, then it is assumed to be a group name and all variables from the the group are exported. Environment variables from groups have the form `M__`*class_name*`__`*variable*.

`class:` will export all groups of class `class`.

The name `Global:` will export all variables from the global namespace.

getenv(env [, variable]) The value of environment variable *env* is assigned to *variable*. If *variable* is not given, the value is assigned to database variable *env*.

include(file [,file2, ...]) Read and process each file.

log(msg) Write *msg* to the log.

run(cmd, ...) *Cmd* is executed and the output is processed as more config commands. Arguments are expanded before calling *cmd*.

setenv(env[, value]) Set an environment value in mio that can be queried by a program executed by the **run** command. *env* is the name of the environmental variable to set. *value* is expaned before assiging to *env*. Only two arguments are allowed. Any additional commas in *value* are treated as part of the value.

If *value* is not given, then the value of the database varible *env* is assigned to the enviromental variable.

tty(msg) Write *msg* to the screen. Arguments are expanded before printing *msg*.

use(name [,...]) Assign variables in group *name* to the current name space. If *name* starts with a `+`, then the variables in group *name* are appened to the current name space.

Variables that begin with an underscore are not assigned.

63.4 Global Variables

Date

Directories A list of directories that contains Package files to be read. This is also used as the default list of packages to load for a Basis code. If the directory name is followed by a `*`, then it will not be include in the load list. A semicolon is used as a barrier in parallel builds in the generated Makefile.

```
Directories = scripts* first ; second third
```

User

AR Defaults to `ar`.

AUXLibs Auxilliary libraries which may be system dependent. These libraries will be put in the load line after the package libraries but before other libraries which `mio` knows are required such as the PACT libraries or the NCAR libraries. These libraries may also be changed for thread safe versions if `mio` knows that it should do so.

default_FGroup

default_CGroup

default_LDGroup

default_LibGroup

default_Mac Set the default Group to use when `Targets` is not defined.

Directories

Glue Defaults to `config`.

INSTALL_MACRO Command to install a file if it does not already exist or the contents have changed. Defaults to `/usr/bin/install -C`.

InstRoot The root directory where the code will be installed.

LD Defaults to `ld`.

NCAR The version of NCAR to use. Options are N4.1 and N4.0 with the default being ”N4.0”.

PackFiles A list of **.pack* files needed for the main executable. The default is no files.

PACTRoot The root directory where PACT is installed. This may also be specified by an environment variable called PACT. The default value will be taken from the environment variable.

Path A blank delimited list of directories which will be added to the beginning of the PATH environment variable when using do-sys to build the application. This allows you to put the location of compilers (or other needed tools) in your config file where you specify which compilers you want to use. This can save you problems with setting up your own environment variables.

POD2MAN Full path of pod2man. A default file is set by searching the current path.

ProgName The name of the principle executable program of the system.

SYSIncPath Include path for headers and other similar kinds of files. This adds additional `-I`*path* to compilations (mppl and cc).

SYSLibs System libraries (usually vendor supplied or installed by the system administrator) used in linking the main executable. These libraries are inserted in the load line last of all and they are taken literally. Compare this with the AUXLibs above. The default is nothing.

SYSLDPath Load path for libraries. This adds additional -C¡-Lpath¿ flags to the load line. You may specify more than one path here. The default is nothing.

63.5 System Group

Mio manages codes as a System. Variables in this group control the location of output files. `Root` is the path to the bin, include, and lib directories.

MakeBin Contains name of variable holding the bin directory path.

MakeInc Contains name of variable holding the include directory path.

MakeLib Contains name of variable holding the lib directory path.

MakeMan

MakeRoot Contains name of variable holding the root directory path.

MakeSrc

VEnv

VMake

VMDef

VCDef

Write_c_defs

Write_f_defs

Write_m_defs

Write_configured

Write_configured_pl

Write_make_config

Write_makefile

Write_mio_csh

Write_mio_make

Write_mio_pl

Write_mio_sh

Write_packages

Write_Packages

63.6 Define Group

Variables in this group are written out as macros.

63.7 Setenv Group

Variables in this group are written out as environmental variables.

VEnv Order to write out variables.

63.8 Compiler Groups

Compiler The compiler executable to use for files in this group.

Debug The compiler options having to do with debugging. This are applied if `-g` options is given to `mio`.

Flags The compiler options that are always passed to the compiler.

Optimize The compiler options having to do with optimization. This are applied if `-o` options is given to `mio`.

Include_path Option to add include search paths. `%s` will be expanded. Typical `-I%name`.

List

List_suffix

Profile

Targets List of files and directories to compile with this group.

Version Option to generate version information

VersionInfo Output from `Compiler`'s `Version` command. Mio runs the compilers for `default_CGroup` and `default_FGroup` to generates this value.

63.9 CGroup Group

Variables for .c files. The global variable `default_CGroup` can be used to set the default CGroup to use to compile.

63.10 FGroup Group

Variabls for .f, .f90 and .m files. The global variable `default_FGroup` can be used to set the default CGroup to use to compile.

MPPLFlags Global MPPL command options.

Glue The name of the program to produce glue file from the `.pack` files. Defaults to `config`.

Module_suffix

Module_path

Module_out

FixedForm Flags to compile fixed form.

FreeForm Flags to compile free form.

Suffix_*suffix* A list of features that will be added to the compile flags for files ending with *suffix*.

63.11 LDGroup Group

Loader options.

Flags Command line options for the linker/loader.

LoadMap

LDpathOpt

LDsearchOpt

MapName If specified, a load map will be created using the value and the command in the `LoadMap` variable.

Profile

63.12 LibGroup Group

For building archives.

ARFlags

LibFlags

63.13 Mac Group

Variables used to control running `mac`.

DocFile Name of generated documentation file. Used with -d option

Flags Global MAC command options.

For expanded values the available values are:

base = base of input file (foo if foo.v). For example `$base_vdf.f90`, with filename foo.v will be expande to foo_vdf.f90

MFile Name of generated macro file. Used with -m option.

WFile Name of generated C file. Used with -w option.

WriteModule The name of the output file for modules. The name is expanded.

YFile Name of generated MPPL file. Used with `mac`'s `-y` option.

63.14 Directory Group

A Directory Group is created and populated with the contents of the **Package** file for each directory listed in the global variable `Directories`.

System System group associated with this package.

PKG = name where `name` is the name of the package. If not given, the name of the directory is used.

ROOT = PKG exe need-root-inst

```
\texttt{PKG} package name as in lib<pkg>.a
\texttt{exe} executable program name (built in this package)
\texttt{need-root-inst} yes | no
```

pkg overrides the package name specified in `PKG`. This is historical because prior to mio, some packages (e.g. rt) had a `PKG` name that was inconsistent with the name of the pkg object or archive and mmm had hard wired code to fix it! *need-root-inst* specifies whether or not the RootInst objects from the global config files are to be copied into the private bin/lib/include directories by this package. The default is nothing.

NeedPACK = use — install — both specify whether *.pack are needed for linking, needed to be installed, or both. Default is nothing.

POINTER = std — cray Specify what kind of Fortran pointers this package uses. The default is `std`.

LIBRARY indicates the default target for this directory is an `ar` library rather than a `.o` file. No effect when making for another machine. Indicates how the library archive is to be built for this package. Without this specification all object files (.o) are preloaded into a single .o file which is placed in the archive. This forces the entire package to be loaded if a single function or variable of the package is referenced elsewhere. With this specification the individual object files are placed in the archive file. This means that only those objects are loaded which resolve a reference generated elsewhere. This choice can have a profound impact on your code system. Be very CAREFUL when deciding which way to go with this variable!!!

MPPL_Flags Flags, included file names, that are added to all uses of mppl in the directory.

MPPL_lang `to f77` Convert the code to f77.

`to f90` Convert the code to f90.

`is f77` Will not process langauge statement but will assume it is already f77.

`is f90` Will not process langauge statement but will assume it is already f90.

ARCHIVE indicates the name of the target library. Defaults to the package name.

VDF = filelist is a list of Basis variable descriptor files.

NVDF = filelist is a list of variable descriptor files which reside in other packages but are needed to compile this one.

SM = filelist Mppl sources which need all VDF and NVDF files.

SU = filelist Mppl sources which need no variable descriptor files.

SF = filelist straight Fortran sources (also supported is the obsolete form `FM`). Files with the suffix `.F` are acceptable but a particular compiler may require setting `FF` (below) to contain a special flag to enable running `/lib/cpp` on the `.F` file before compiling.

SC = filelist C or C++ sources. Names of header files on which the sources depend should be placed in the list in front of the files on which they depend.

CLEAR Used to "forget" any previous dependencies. For example, suppose `foo.c` depends on `foo.h`, and `bar.c` depends on `bar.h` but *not* `foo.h`. This would be denoted as follows in the `Package` file:

```
SC=foo.h foo.c
CLEAR
SC=bar.h bar.c
```

Without the reserved word `CLEAR`, `mio` would think that `bar.c` also depended on `foo.h`, and build makefiles accordingly; the result would be that an unnecessary compilation of `bar.c` would occur every time `foo.h` was changed.

LANGUAGE = langlist `langlist` can consist of one or more of `C`, `C++`, or `FORTRAN`, `FORTRAN` being the default. This statement must precede the declaration of any list of VDF's or NVDF's which contain `language "C"` or `language "C++"` statements, so that `mio` will be able to build appropriate makefiles. If there is a later list of VDF's and/or NVDF's not containing C or C++, then `LANGUAGE=FORTRAN` will keep `mio` from making unnecessary C or C++-specific makefiles.

SENDFILES = filelist files to be sent via `ftp` if `make` is done for a remote machine

FF = line The generated makefile will define the Fortran compiler flag `FF` to be the rest of this line. The default is a possibly acceptable set for a given CPU. See further discussion in the section COMPILER FLAGS in the manual page.

CF = line The generated makefile will define the C compiler flag `CF` to be the rest of this line. The default is a pretty good set for a given CPU. See further discussion in the section COMPILER FLAGS in the manual page. Note that `CF` should generally not be used for optimization flags; see the section OPTIMIZATION.

Real4 Sets the default meaning of a Fortran real to be 4-byte

Real8 Sets the default meaning of a Fortran real to be 8-byte

RULE/ENDRULE Text between RULE and ENDRULE is copied literally into the pre-Make file. This allows you to manage targets and control dependencies explicitly if the automatically supplied rules do not suffice.

A line containing SYSTEM followed by one or more of the architecture names will cause subsequent lines to be ignored unless the name of the target CPU is one of the set. This SYSTEM directive works the same as it does in mac and gluepack, which were described in earlier chapters. For example:

```
PKG=foo
VDF=foo.v
SM=always.m
metoo.m youtoo.m
SYSTEM SUN4 HP700
SM=workstation.m
SYSTEM YMP
SM=unicos.m
```

The file workstation.m will be used as a source if CPU=SUN4 or HP700. The files foo.v, always.m, metoo.m and youtoo.m are used on all platforms.

You can also do differential compilation within an MPPL-language file using constructs of this type:

```
ifelse(SYSTEM,HP700,[
       ...code for HP700 only
])
ifelse(SYSTEM,YMP|XMP,[
       ...for XMP or YMP
])
ifelse(WORDSIZE,32,[
       ...code for 32 bit machines
])
```

You execute mio by executing BASIS_ROOT/bin/mio. To build a debuggable code, add the -g option. If you wish to link ith a profiler, use the -pro option. After this, the command make should cause your packages to be compiled. mio will create a subdirectory ARCH where ARCH is uniquely identified of the system you are running on, such as osf-5.1, lnx-2.2-i32, or sol-5.2. All the output from the make will be in this ARCH subdirectory. **NOTE:** these ARCH names are generated by the Basis txtttcpu that uses the UNIX uname commandto generate unique platform and system-dependent names. Once this is successful, proceed to the next section.

63.15 File Group

Dependencies Build dependencies of file.

Module Modules generated by file.

Phase Name of phase to compile file `mac` or `build`. Defaults to `build`.

MPPL_Flags Flags to pass to MPPL. If not defined then the generated pre-Make file will use `$MPPL_flags`.

63.16 Package Group

System System group associated with this package.

63.17 Archive Group

63.18 Library Group

63.19 Program Group

BinDir Directory for final executable. Setting `BinDir` to blank will leave executable in the *Arch* directory below the package directory. If `BinDir` is not set, executable will be in `$(SysBin)`.

LDFlags Additional loader flags

LibPaths Library search paths to use.

Libs Library to use.

MapName Works with the LDGroup's *LoadMap* options.

Source List of source/object files used to build program.

63.20 BasisProgram Group

DocFile

GlueFlags

Name Name of executable. Defaults to the group name.

LDFlags Additional loader flags

LibPaths Library search paths to use.

Libs Library to use.

Main 1 = load with Basis' main program. defaults to 1.

Packages List of Package and Library groups to use. Defaults to the `$Directories`. '`par`' is always appended to the end.

PackFiles

PackBase Name of generated pack file without any suffix. Defaults to '`pack`' appended to the executable name.

Phase

63.21 Fparse Group

Flags

GenerateInterface Options are *no*, *mppl*

MPPLInterface Generate macros to use the mppl interface blocks from other packages. Writes file `mio_`*dir*`.d`.

Valid values are B¡mppl¿, B¡module¿, B¡include¿, B¡no¿.

Modules List of modules to parse before source. Added as a `--module name` option to fparse.

RunIface If set to 'no', turns of running fparse.

CHAPTER

SIXTYFOUR

Getting Started Writing Packages

If your goal is to quickly make a program for the purpose of executing one or two functions interactively, you can do that without reading this manual in full. There is a program called `basiskit.` Make sure you have your environment and path set up as described in Section 1.1 Environment Variables in Chapter 1. Create an empty directory and in it type:

```
basiskit cbk
```

This will create a source file cbk.m which you replace with your own. Edit cbk.v to describe your own common blocks and variables instead of the sample ones. If you have a common block labeled /xyz/ that you wish to link to the interpreter, declare a group (like the one Variables in the sample), and after the group name put /xyz/ before the colon. Then describe the common block variables in exactly the order in which they occur in your source.

If your source does not already exist you can edit `cbk.m` instead. Follow the instructions `basiskit` printed out.

The following sections describe the components you will be working with.

64.1 Outline of the Process

Producing a program under the Basis system is very easy. In addition to your sources, you need to create a small number of input files to the various Basis utilities, then run the utility `mio` ("**m**ake **i**s **ok**") which creates makefiles that, when processed by the unix `make` utility, control the execution of the other Basis utilities and build your code automatically. Basis goes one step further and provides a model `dsys` for managing building and testing your code across multiple platforms or operating systems. `Dsys` is decribed in chapter 4. We describe here the key elements of building a Basis code application.

The basic outline of the directory structure will serve to clarify the following discussion of the `dsys` utility.

```
mycode/
       source code for mycode
       mycode.v
       mycode.pack
       Package
       builder/
              dsys
              local/
                    config-file-platform1
                    config-file-platform2 ...
              std/
                  packages ...
```

At the top-level source directory tree for `mycode`, you will see:

- In the file "`mycode.v`", you declare your variables in a separate file called a variable description file or VDF. You divide these variables into named groups similar to named common blocks. You also declare those subroutines and functions you wish to be able to call interactively at run-time. The VDF can be likened to a C or C++ header file, in that you can replicate all or portions of its data declarations in your code. The VDF is processed by the Basis utility `mac`.
- In the "`mycode.pack`" file, you declare overall configuration and "packaging" information about your application (`mycode`).
- In the "`Package`" file you define the VDF files to be included, the names of the source files to be compiled and the language the source files are written in.
- A `builder` subdirectory.

The `builder` subdirectory contains the `dsys` utility, related utilities (for more advanced functions, such as automated testing, that we won't go into here), plus it's own subdirectories of platform-dependent configuration `config` files. The files most critical to the build process are:

- The `dsys` utility. This utility orchestrates the procedure which creates the makefiles and turns your sources into compilable modules per-platform. `Dsys` runs `mio` and other Basis utilities to crreate your compilable source, and puts them into uniquely named platform subdirectories of your source directory `mycode`.
- In the `local` subdirectory, you provide per-platform customization information such as compiler options or language feature options in a per-platform file.
- In the `std` subdirectory, you may specify the lowest-level feature-independent elements common to a particular platform. For instance, in "`package`" you would specify which standard Basis packages you wish to include.

These files are all that you need to create (other than your sources) if you wish to have your application built automatically.

Your source files can be in Fortran, C, C++, or MPPL. MPPL is an upward compatible extension of Fortran 77 that comes with the Basis System. The preprocessor `mppl` takes MPPL language input and produces standard Fortran output. Existing routines can be used with Basis with little or no change. However, most Basis authors use `mppl` and many of the optional services described later. Using `mppl`, for example, you need only maintain the list of common variables in the variable description file. In your `mppl` source, you put the statement:

```
Use(Groupname)
```

in each subroutine that needs the variables in the group named `Groupname`. "`Use`" is an `mppl` macro which expands into the correct common block declarations for the group in question.

A program called `gluepack` writes a small set of routines that connect your source package to the routines supplied with Basis. These latter routines include a main program and the Basis Language interpreter.

In the compilation process, a program named `mac` processes the variable description file into a macro file and a file of special subroutines; these files, together with your sources and the output of `gluepack`, are then preprocessed by the program `mppl` into standard Fortran source files that you compile with `f77`, `f90`, `C` or `C++`.

Finally, load your program with a binary library called `libbasis.a` that contains the Basis system run-time routines.

In practice, the utility *mio* is used to generate input files for the Unix utility *make* and you don't actually run config, mac, mppl, or the compiler/loader yourself.

A Basis program consists of one or more Basis packages, so the first thing to know is how to make a Basis package. Then the construction of the whole program will be covered.

CHAPTER

SIXTYFIVE

A Complete Example

65.1 Overview

The following is an example of using Basis to do algorithm development. In FORTRAN, we write the algorithm we are working on so that we can execute it by calling the following function, which we put in a file `wve.m`:

```
      subroutine xyz(alpha,beta)
Use(Vars)
      .... algorithm goes here ....
      return
c come here if something goes wrong
900   call remark("xyz: algorithm failed.")
      call kaboom(0)
      end
```

The idea is that group `Vars` will contain all the data structures needed to set up the problem. Our Basis Language input file will contain statements to set up the initial values, a call to `xyz`, and then statements to print or plot the results.

65.2 Variable Description File

This file `wve.v` declares the parameters `nz` and `nt` to set the size of a mesh, and then some derived sizes `neq`, `nb`, `nbf`. It contains one group named `Vars` which contains 8 variables `phi`, `phib`, `dz`, `dt`, `v`, `tau`, `cin`, and `cout`, and two scratch arrays `a` and `b`. To test the algorithm we will set values of `phib`, `dz`, `dt`, `v`, and `tau`, and then call `xyz` with test arguments `alpha` and `beta`. The results will be in `phi`, `cin`, and `cout`. Default values for `dz`, `dt`, `v`, and `tau` are data-loaded.

```
wve
{
```

```
nz=100 # number of zones
nt=100 # number of timesteps
neq=nz*nt
nb=nz+1
nbf=2*nb+1
}
******* Vars:
phi(nz,nt) [Number/cm] # number density
phib(nz,nt) [Number/cm] # boundary condition
dz /1./ [cm]
dt /1./ [sec]
v  /3.14159/ [cm/sec]
tau /1./ [sec]
cin [Number] # number in
cout [Number] # number out
a(neq,nbf) real  #work space needed by algorithm
b(neq+nbf) real  #work space needed by algorithm
xyz(alpha, beta) subroutine
   #This declaration lets Basis know how to call xyz
```

65.3 config input File

This file, `Configure`, besides declaring the wve package, causes Basis to initialize wve immediately on startup and personalizes the code name and prompt.

```
package  wve = "Test my algorithm"
firstpkg = wve
codename = "Wave"
cprompt = "Wave> "
```

65.4 mio input Files

At this point we have prepared three files: `wve.v`, `wve.m`, and `Configure`. To make the program using `mio`, we first need to run `mio` after preparing its input files.

The `mmm` input files are pretty simple.

WHAT GOES HERE

That tells `mio` that we wish to make a program, not just a package, in this directory. Second, we prepare the `Package` file:

```
PKG wve
SM=wve.m
```

```
VDF=wve.v
Real4  #let reals be 32 bit on workstations
```

Typically we would use the `-g` option while debugging:

```
   BASIS_ROOT/bin/mio -g
```

Recall that `BASIS_ROOT` here stands for the directory holding the Basis distribution. We might also have used the `-nog` option to `mmm` to load the program without graphics, or `-V` to produce verbose makefiles.

65.5 Compiling and Loading

Let us assume the system is HP700.

```
make all code
```

`mmm` will have created an `HP700` subdirectory into which all the output of the compile/load process is placed. `mppl` errors in precompiling `wve.m`, for example, can be found in `HP700/wve.f.err`. Compiler errors in compiling `wve.f` will be found in `wve.err`. The program itself is in `HP700/code`, any load errors in `HP700/code.err`, and a load map is in `HP700/code.map`.

When the program is run the input can be interactive, or, in this example, in a file.

```
HP700/code read myprob / 5 6
```

where `myprob` is a file containing the Basis commands:

```
integer i,nz=100,nt=100
# boundary conditions
   do i=1,nz
    phib(i,1)=exp(-4.*(i-1.)**2/(nz-1.)**2)
   enddo
   do i=1,nt
    phib(1,i)=exp(-4.*(i-1.)**2/(nt-1.)**2)
   enddo
# try calling xyz
   call xyz(1., 2.)
# make EZN contour plot of phi
   plot phi, iota(nz), iota(nt)
   end
```

Without the `END` statement, control would return to the terminal after the statements in `myprob` had been processed.

65.6 Changing to Dynamic Memory

We used parameters `nz` and `nt` to set the size of the mesh. It is nicer to use dynamic memory so that these sizes can be changed at will. The main changes are to the variable descriptor file:

```
wve
******* Vars:
nz /100/  #number of zones
nt /100/  #number of timesteps
neq  #set in generate
nb   #set in generate
nbf   #set in generate
phi(nz,nt)  _real [Number/cm] # number density
phib(nz,nt) _real [Number/cm] # boundary condition
dz /1./ [cm]
dt /1./ [sec]
v  /3.14159/ [cm/sec]
tau /1./ [sec]
cin [Number] # number in
cout [Number] # number out
a(neq,nbf) _real
b(neq+nbf) _real
xyz(alpha, beta) subroutine
   #This declaration lets Basis know how to call xyz
makeroom subroutine
   # This routine allocates space for everything.
```

This file declares dynamic arrays `phi`, `phib`, `a`, and `b`. The algorithm to be tested requires boundary values in the array `phib`. The idea is to read the input, which gives values for `nz` and `nt`, calls `makeroom` to allocate space for the dynamic arrays, computes values for `phib`, and then takes one step to calculate the answer.

To do this, we have made `nz`, `nt`, `neq`, `nb`, and `nbf` into variables, and put underscores in front of the types of `phi`, `phib`, `a`, and `b`. We add to our `mppl` source file `wve.m` a new subroutine `makeroom` to allocate the memory using the Basis facility `gallot`, and add a description of makeroom to the variable descriptor file as shown above.

```
      integer function makeroom
Use(Vars)
      integer gallot
      external gallot
      neq=nz*nt
      nb=nz+1
      nbf=2*nb+1
      if( gallot("wve.Vars",0) = ERR) return(ERR)
```

```
      return(OK)
      end
```

We change our input file to set values for `nz` and `nt`, call `makeroom` to allocate storage, then set the values of `phib`, call `xyz`, and finally plot the result as before.

```
   integer i
# set desired nz and nt, then allocate space
   nz = 50
   nt = 60
   makeroom
# boundary conditions
   do i=1,nz
     phib(i,1)=exp(-4.*(i-1.)**2/(nz-1.)**2)
   enddo
   do i=1,nt
     phib(1,i)=exp(-4.*(i-1.)**2/(nt-1.)**2)
   enddo
# run problem
   call xyz(1., 2.)
# make contour map of phi
   plot phi,iota(nz),iota(nt)
   end
```

CHAPTER

SIXTYSIX

Compiling Basis Packages

Once you have constructed a variable description file and a source file, you are almost ready to compile and load with the Basis run-time system.

You need to know where your Basis distribution is. Frequently, it is in

```
/usr/local/apps/basis (on LC systems)
```

In what follows, we will refer to this directory as `BASIS_ROOT`.

Other files of importance include:

BASIS_ROOT/bin contains Basis executables, and may be added to your path.

BASIS_ROOT/man contains Basis manual pages, and may be added to your `MANPATH`.

BASIS_ROOT/lib contains Basis's binary libraries

66.1 Single Package Example

For starting purposes suppose you have a non-Basis code which consists of three files, a.f, b.c, and c.h in a directory /foo. So if you

```
cd ~/foo
ls
```

you will see:

```
a.f   b.c   c.h
```

When you compile and link your code you get an executable called foo.

The steps to turn this in a Basis code are:

1. Write a VDF
2. Setup the configuration management
   (a) Setup the builder directory
   (b) Write a config file
   (c) Write the Package file
   (d) Write the Configure file
   (e) Run mio
3. Build the code
4. Making changes

Write a VDF Following the outline in the chapter "Writing Basis Packages", write the variable definition file which will be a part of the interface of your code to Basis. For the rest of this example it will be assumed to be called foo.v.

Convert .f files to .m files Following the outline in the chapter "Writing Basis Packages", convert your Fortran files, .f to their .m counterpart. Many times, this is a simple matter of replacing your common blocks with analogous statements in a VDF file, and renaming the remaining Fortran file. This makes connections between your routines and the Basis interpreter and other Basis facilities. In our example then there will be a.m instead of a.f.

Setup the configuration management A Basis code is structured in such a way that it can take advantage of the various services which Basis offers. It is also structured to be portable and to easily support building on many, different hardware platforms simultaneously.

Setup the builder directory Make a directory called **builder** and go to it. In the builder directory you will keep your config files and any scripts you want to use to control compilations, testing, installations, and so on.

Write a config file Following the section of the mio man page about global config files write one or more config files for your code. You will probably want to have at least one config file per hardware/os platform you build on. You may want tohave different config files to build versions of your code with profiling or alternative feature sets.

Here is an example which might be apropriate for your codewhen built on a Linux box. Let's call this config file "lnx".

```
#
# LINUX - basic LINUX Basis configuration
#

ProgName  = foo
```

```
Packages  = .

Packages  = .
PackFiles = ${BasInc}/ezn.pack
RootInst  =

AUXLibs = -lezn
SYSLibs =

FGroup : 1 {
  use(pgi_f90)
  Flags    = -Mrecursive
  Optimize = -O2
}

CGroup : 1 {
  use(gnu_cc)
  Flags    = -Wall
  Optimize = -O3
}
```

In this config file the code will be named foo and it will be using the Basis EZN graphics package.

66.1.1 Write the Package file

In your packag’e directory

```
~/foo
```

you will write a `Package` config file for you code.

Here is a `Package` file which might be a start for the files described at the beginning:

```
PKG = foo
SC = b.c
SM = a.m
VDF = foo.v
```

Notice the references to the files, a.m and foo.v mentioned earlier.

66.1.2 Write the Configure file

Next you will want to write a file called `Configure` which will give Basis some details of how you want your code to be linked and how it will look at run time.

Here is a simple file for our foo code, which sets the banner for the code start up and the code prompt:

```
codename = Foo
cprompt = "FOO> "
```

66.1.3 Run mio

At this point you can actuall run `mio` to ”configure” your code system. You are not yet ready to actually compile anything, but you can have mio do its jobs and be ready to compile.

```
cd builder
```

Run `mio` in the builder subdirectory to get the entire system configured to build.

```
mio -a lnx
```

Here we told mio to use the lnx config file described earlier. When mio completes there should be a set of directories

```
~/foo/dev/linux/lib
~/foo/dev/linux/bin
~/foo/dev/linux/include
~/foo/dev/linux/log
~/foo/dev/linux/man/man1
```

These contain files that you will build/install in the next step.

66.1.4 Build the code

Now we are ready to compile and link the code. When `mio` finished it left a script in each directory mentioned inthe config file (in this case in lnx). So now do the following:

```
cd ..
pck build
```

When this is done you should find an executabl e file called foo (which is what was specified) in the bin directory. That is

```
~/foo/dev/linux/bin/foo
```

should be the executable. You should also see an archive file

```
~/foo/dev/linux/lib/libfoo.a
```

which contains a.o, b.o, and foo.y.o.

66.1.5 Making changes

As you develop your code you will make changes. To recompile and relink just do:

```
pck build
```

If you want to do a clean, from scratch, build do the following:

```
pck remove
pck build
```

This should get you started Consult the other Basis documentation for more details on the individual pieces mentioned here.

66.2 Adding a Second Package

Suppose your code now grows and you want to reorganize it into two or more packages. Suppose packages a and b are made. Package a contains foo.v, a.m, b.m, c.m and package b contains x.c and y.c.

66.2.1 Reorganize the directory structure

Before foo contained

```
foo/Configure
foo/Package
foo/a.m, foo.v, b.c, c.h
foo/builder/
foo/dev/
```

Now make directories for both packages and move files around so that foo looks like:

```
foo/a a
foo/b
foo/builder
foo/dev
```

where

```
ls ~/foo/a
```

gives

```
Configure   Package  a.m    b.m   c.m
```

and

```
ls ~/foo/b
```

gives

```
Package   c.h   x.c   y.c
```

66.2.2 (Re)Write the Package files

You will write Package files appropriately for each package. The `Configure` file goes with package a and the main executable foo will be built there.

66.2.3 Modify the config files

Now you have to change the config files you have. For example the lnx config file becomes:

```
#
# LINUX - basic LINUX Basis configuration
#

ProgName  = foo
Packages  = b a
PackFiles = ${BasInc}/ezn.pack
RootInst  =

AUXLibs = -lezn
SYSLibs =

FGroup : 1 {
  use(pgi_f90)
  Flags    = -Mrecursive
  Optimize = -O2
}

CGroup : 1 {
  use(gnu_cc)
  Flags    = -Wall
  Optimize = -O3
}
```

Notice the change in the Packages specification. Also notice the order. `Package` is the one in which the link step will be done so it must come last.

66.2.4 Reconfigure

Run `mio` to reconfigure all the packages as well as the main code.

```
cd ~/foo/builder
mio -a lnx
```

66.2.5 Rebuild

Now to build the code do the following:

```
cd ../b
pck build
cd ../a
pck build
```

CHAPTER

SIXTYSEVEN

Writing Basis Packages

67.1 Basis Packages

A Basis package is a set of modules that perform some calculation. A program consists of one or more packages together with the Basis run-time routines. This chapter explains how to write a Basis package. You will learn how to write a variable descriptor file, in which you describe your variables and functions so that Basis can access them, and how to write your source.

The Basis Package Library includes a package `ctl` implementing a simple generate-step-finish model, which you may use or not as you wish. If you do not include `ctl` as a package in your program, you will need to list the functions you wish to be able to execute in your variable description file.

To begin a new package, select a two- or three-letter lower case package name. We use `pkg` in the examples in this manual. The length of a package name is limited to three characters. (This limit is a consequence of the historical limits on the lengths of Fortran external names.) As many as 75 packages can be loaded together into a single code.

To avoid conflicts with the standard packages available with Basis, do not use these names for your package:

```
par, rt, bgr, bdp, pgs, edt, ezn, ezd, rng,
bes, ctl, fft, fit, hst, pfb, svd, tim
```

CHAPTER

SIXTYEIGHT

Precision and Portability

68.1 Description of the Problem

Precision problems arise for a number of reasons. For one thing, FORTRAN's implicit typing (variables beginning with i-n are integers, all others are reals) has created a couple of generations of programmers who have not acquired the laudable habit of declaring all variables. This perhaps might not be such a big problem were it not for the fact that a real, for instance, is sometimes 32 bits long and sometimes 64 bits long. Thus even those individuals who declare all variables will have inconsistent results from one platform to another. For portability and consistency of results among different platforms, it would be nice if reals were always the same length.

Another problem can be caused by using intrinsic function names that are specific to certain types of arguments and results, rather than generic, e. g., `MIN0` (`integer`), `AMIN1` (`real`), and `DMIN1` (`double precision`). Even if you could force all reals to be 64 bits long (say), a code still might contain calls to `AMIN1` rather than the generic `MIN`, which would cause loss of significance or an argument type mismatch on 32 bit machines.

Fortunately the Basis team has provided solutions for these headaches.

68.2 Specifying Precision in the Source

`mppl` accepts an option `-r8` which causes it to produce standard Fortran output in which the default meaning of the type `real` will be either `real` or `doubleprecision` (depending on architecture), so that the result is guaranteed to be a 64-bit quantity. The Fortran 90-like kind-selector syntax `real(Size4)` can then be used to force a 32-bit quantity where desired, assuming that 32-bit reals are available on the target architecture. Likewise, `mppl` makes the default type of literal real constants 64-bit, and the syntax `1.0_Size4` can be used to override this. Full details are available in the `mppl` man pages.

By default, `mio` uses the `-r8` option on `mppl` input files. To determine the `mppl` option yourself, add a line with the word `Real4` or `Real8` to the `Package` file.

For random numbers use the `ranf()` function, which produces an identical random number stream on all platforms.

Variables which are not declared are implicitly typed `real` by the Fortran compiler if they have names beginning with the letters `a-h`, `o-z`. `mppl` will not declare such variables `double precision` where appropriate, leading to a loss of precision in expressions or to function parameter/argument mismatches. You must either declare all variables, or insert the statements:

```
implicit integer(i-n)
implicit real(a-h,o-z)
```

into each routine with undeclared `real` variables. A compiler flag is often available to detect undeclared variables. In lieu of inserting such statements, you may wish to use the `mppl Prologue` macro which you can define to be the statements above.

68.3 Making Your Source Portable

Given a source file `foo`,

```
BASIS_ROOT/bin/generify foo >bar
```

produces file `bar` in which each Fortran intrinsic function reference has been replaced by its generic form, such as changing `amin1` to `min`. Without such changes, a loss of precision will result when using the `-r8` facility.

The default interpretation of an argument to a function as described in a variable description file is that an untyped name is typed `integer` or `real` by the Fortran naming convention. In a program in which the function is being compiled on a 32-bit machine under the influence of the `-r8` option to `mppl`, a function argument implicitly typed `real` will be 32 bits long. on the other hand, when the variable descriptor file is processed, an error would occur in the default case, because a variable either implicitly or explicitly declared real becomes `double precision`. So, be sure to explicitly type such function arguments (and results) `real`, both in the VDF and in FORTRAN, as in:

```
foo(x:real, y:complex, z:integer) real function # in the VDF
```

and in the FORTRAN

```
    real function foo (x, y, z)
    real x
    complex y
    integer z
```

`mppl` will take care of ensuring that both instances of `x` will be the same length, which would not have been the case if `x` had not been explicitly declared. `f`, too, had to be explicitly declared in this example.

It is worth repeating that if what you want is really a 32-bit real (and your architecture supports it) then you need to declare it `real(Size4)`. If you want all of your reals to be 32 bits, the easiest thing is to put the `Real4` statement in your `Package` file, run `mio` (it will produce `mppl` rules with the `-r4` option), then do a `make clean` followed by a `make all code`.

CHAPTER

SIXTYNINE

Fcc: Fortran Calls C

CHAPTER

SEVENTY

Mac and the Variable Description File

A variable description file describes common block variables and Fortran subroutines and functions. The Basis System routine `mac` converts this file into routines which describe the variables and functions to the Basis Language interpreter. These routines are called when your program initializes, and enter the variables and functions in the Basis database, together with important information such as type, dimensions, etc. So, after describing a variable named `x` in a variable description file, when the resulting program executes, `x` can be used interactively in Basis Language statements. You can also describe variables which are used in MPPL source files but not known to the Basis Language (so-called hidden variables). Furthermore, the comments in the variable description file may be retrieved at run-time.

70.1 Sample Variable Description File

The following sample file may be enough to allow you to write your variable description file without reading the rest of this section in detail. You write a separate variable description file for each package you create.

```
pkg
# this is a comment about the pkg package
# more comments go here (such as revision history);
# next comes the parameters enclosed in a set of optional braces,
# then the first group which we name Geometry
# the second group called Switches, and the third, Routines.
{
My_parameter = 2200
N = 10
NP1 = N + 1
}

***** Geometry:
#Variables which describe the geometry of the machine
```

```
x(N)    real [cm]  /N*0./  # x holds lengths of boards


xlength [m]  /42.0/
#length of machine, defaults to 42 meters

ws(My_parameter)                    #workspace

bigws(1)     _real            # dynamic work space
                              # gets allocated in generator
******* Switches:
# option switches
nails   integer  /NO/          # YES means use nails, not string
gamma   integer /YES/          # YES means include gamma rays
****** Routines:
# Fortran routines we can call
alpha(a:integer, b:real) subroutine # sets model parameters
integ(f:external, a:real, b:real) real function
   #integrate f from a to b
```

70.2 Structure of the File

The variable description file consists of a header followed by the description of one or more groups. The header contains, in the following order, optional comments, the name of the package, optional comments, and an optional section that defines symbolic parameters. The parameter section may be enclosed in braces as above, but this is not required as it was in older versions. Each group consists of a group information line, comments about the group, and then a series of one or more variable declarations.

70.3 Parameters

If desired, you can define symbolic constants to be used in your package after the package name. These parameters are typically constants or sizes of things. Parameters are global to all modules in the package defined by the variable descriptor file, more analogous to C macros defined in header files, rather than FORTRAN parameters. (Again we emphasize that the braces are optional, but we shall include them in all of our examples.) The syntax is:

```
{
parameterlist
}
```

where `parameterlist` consists of a series of comma- or blank-separated parameter definitions, of the form

```
Parameter_name1 = value1, Parameter_name2 = value2
```

or

```
define Parameter_name value
```

Parameter names must begin with a letter and can be 1 to 32 characters long and include underscores. Parameter values can be integer, real, octal, or hexadecimal constants, strings quoted in either double or single quotes, or anything enclosed in square brackets. An octal constant is an integer constant followed by the letter ‘b’. A hexadecimal constant is an integer constant followed by the letter ‘x’.

Parameter values can also be defined using arithmetic expressions that contain constants and names and the operators plus(+), minus(-), multiply(*), divide(/), and exponentiate (**). In addition, the operators `integer`, `real`, and `character` are available to coerce types. The coercion operators have the highest precedence. Here are some examples of parameter definitions.

```
{
NMAX = 32                   #you can include comments
NMAX_plus_1 = NMAX + 1
NMAX2 = NMAX/2
Root2 = 1.414159            #sqrt(2.)
Root2_over_2 = (Root2)/2.
Greeting = "Hello World"
Reply = 'Get Lost'
define MAXCASE 400
define Prologue [implicit automatic(none)]
One = integer Root2  #Result is 1
FNMAX = real NMAX  #Result is 32.0
SEVEN = 7
HALFSEVEN = real 7 / 2 #Result is 3.5
THREE = integer( real 7 / 2) # (or just 7/2)
WORDS = character Root2 + 1 #Result: string "1.414159e00+1"
}
```

Basis will calculate the value of each parameter. (If a parameter expression involves a name which is not a previously defined parameter, the parameter will be defined only in string form; no warning message will be issued. `mac` assumes that the name will be resolved later by `mppl`.) The result of a parameter evaluation will be integer if all components are integer, or real if any component is real or if any component is raised to a negative power, integer or not. You may use the parameters subsequently in the variable description file and in your source program.

Any line in the parameter section, or indeed, anywhere in the variable descriptor file, that begins with a percent sign (%), is copied directly into the `mac` output file `macpkg` with the percent sign

removed. This enables you to insert complicated macros whose evaluation will be performed by `mppl`, or to insert regular FORTRAN statements into your groups.

The parameter section also has a limited facility for declaring user-defined types. There is a section later in the chapter describing this feature.

70.4 Group Information

Divide your variables into sets, called groups. A group should contain variables that are often used together or that belong together in natural ways, such as those describing some physical state.

A new group in the description file begins with a line containing three or more asterisks, or the reserved word `Group`, or both, followed by the name of the group, then an optional series of one or more words that describe the attributes of the group, and ending in a colon. The general form is,

```
**** Groupname scope attributelist:
# comments
```

`Groupname` must be alphanumeric and begin with an upper-case letter. Underscores may be used after the first character. The name is followed by an optional scope declaration and an optional arbitrary list of words called "attributes". Finally, the list of scope and attribute words is terminated with a colon. The list may extend over several lines and be separated by blanks or commas.

The group description can include one or more comments. A comment is everything from a pound sign (#) or dollar sign ($) to the end of the line. Normal comments begin with a pound sign; comments that begin with a dollar sign are not output by any of the programs that access the description file, and are thus private remarks.

70.4.1 Scope

If no scope word is specified, Basis creates common block names for the variables in this group. The variables are "visible", that is, they will be known interactively to the Basis Language at run-time. These defaults can be changed by declaring the group `local`, or by specifying the common block name explicitly. Using the word `hidden` hides the variables from the Basis run time system. Here are the details of these choices:

`local` designates that this group of variables is local to the subroutine in which they are used. They are not placed in common blocks, so the same group `Use`'d in another subroutine will not be the same variables. This group of variables is declared in a subroutine using the `Use` macro in the usual way, but local variables do not get entered into the run-time database manager; they are not known to Basis at run-time.

`/label/` designates a label for the common block to be generated for this group. The label name is enclosed in slashes. */label/* allows you to declare common blocks that are used in software

supplied by others, such as mathematical packages. See, however, the cautions on labeled common below.

`hidden` The word *hidden* makes this group of variables unknown to the run-time system. The user will not be able to set or display any variables in the group interactively. The word hidden may be combined with a common block label.

`compileas(spec)` *(spec)* is used to give the compiler and Basis differing views of the dimensioning of the dynamic variables in a group. This feature is described below since it is essentially a variable description. It is allowed as part of the group header to indicate that it applies to all variables in the group.

`language "LANG"` This specification tells Basis that you want to access the variables in this group from code written in the language "LANG" (which can be C, C++, or FORTRAN– which is the default, of course). If you specify C or C++, Basis will create so-called "glue" code which will allow C or C++ code to access these variables. To access functions written in C or C++ from FORTRAN code or Basis, put their descriptions in groups with the "`language "C"`" or "`language "C++"`" specification. Please refer to the man pages for `mac`, `mio`, and `Fcc` for complete details.

When you use `/label/`:

1. Do not mix character and non-character variables in the same common block. This is not Fortran standard, even though older compilers, like `f77` support the mixture.

2. Be sure that the variables are listed in the group in the exact order in which they are to appear in the common block.

3. With certain compilers, and on some 64-bit architectures, alignment problems are likely to arise if variables of different size are declared in the same common block. If you do not specify `/label/`, Basis will create separate named common blocks for variables of different type, thus eliminating this problem.

70.4.2 Attributes

You may specify "attributes" for variables. An attribute is simply a word, beginning with an upper- or lower-case letter, including digits, underscores, and letters, up to 24 characters in length. An attribute declared for a group applies to all the variables in that group (unless overridden in the description of the variable). The Basis `LIST` command lists the attributes of a variable. The subroutine `rtcattr` can be used to change the attributes of a variable at run-time. The routine `rtattr` tests whether or not a variable has a given attribute.

Thus, attribute words at the simplest level can be used simply as documentation. Basis has facilities which make it easy to do something to all variables having a given attribute. Four such routines are supplied:

- `attredit(jout,attribute)` writes the values of every variable having the given `attribute`, onto the file connected to unit `jout`.
- `attrlist(jout,attribute)` lists every variable with `attribute`.
- `rtattr(name, attribute)` returns `TRUE` if name has `attribute`.
- `rtcattr(name,attrstring)` changes the attributes of name as specified by `attrstring`.

Section [Ref: wrattrser] "Writing Attribute Services" discusses how to write similar facilities of your own.

70.5 Variable Descriptions

Following the group description line and any comment lines, you can declare one or more variables. The name of each variable must begin with a lower-case letter. A variable description begins with the name of the variable followed by optional information about the dimension, type, units, initial value, and attributes in any order:

```
variablename(dimension) type [units] /initialvalue/
              +attribute -attribute "varname"   #comment
```

where

`(dimension)` is a dimension for the variable, enclosed in parentheses, e.g., `(100)` produces an array of size 100. (See further discussion in section [Ref: dynamic-dimensioning] "Dynamic Dimensioning.")

`type` specifies the type of the variable, using a Fortran type such as real, double or complex, or a symbolic type. An underscore preceding the type name (`_type`) declares a pointered variable of that type (see section [Ref: dynamic-dimensioning] "Dynamic Dimensioning.") If the type is omitted, it is inferred from `variablename` integer if the name begins with `i` through `n` inclusive, real otherwise. (Note: Cray's `CFT77` does not presently handle dynamic character arrays.)

`[units]` gives the units of the data contained in this variable, enclosed in square brackets. For example, `[cm]` means this variable contains data in centimeters. This information is used for documentation and labeling purposes only.

`/initialvalue/` is a Fortran data specification, as in `/4.333/`. The user encloses `initialvalue` in slashes as shown. Initializations that are awkward or impossible to handle in this way should be done in subroutine `pkginit`. If the variable is dynamic, the data specification if present must be a scalar and must be of the correct type. This value is then used by `allot` and `change` to initialize the variable's contents when space is obtained for it.

+attribute gives the variable the attribute whose name follows the plus sign. If the group to which the variable belongs has a certain attribute the variable has that attribute by default. This feature allows you to give a variable an attribute in addition to any that it inherits from the group.

-attribute removes the attribute whose name follows the minus sign. This allows you to give a group a certain attribute but exclude some members of the group.

"varname" The `"varname"` generates an equivalence statement between '`variablename`' and '`varname`'.

There are some other special keywords which can be added to a variable description: `limited`, `compileas`, `function`, `subroutine`, and `builtin`. These are discussed next.

70.6 Limiting Array Sizes

limited(dimension) Authors may add the keyword `limited` to the description of any variable in the variable description file. This will cause a call to `setlimit`, described below, to be made when the package is initialized. The effect of this call is to cause the length of a variable to be recalculated whenever the variable is referenced by Basis.

The keyword `limited` may be followed by a dimensioning string. This will be the string passed to `setlimit`. If such a string is not given, the dimension string for the variable will be used.

Here is an example of using the `limited` keyword:

```
n           integer
      # this integer controls the lengths of x and y
x(100)        limited(n)
      # x behaves at all times as if n long;
      # error if n>100, but Basis does not check this.
y(n)           limited       _real
      #y dynamic, behaves as if n long (but allot/change
      #knows the actual size)
```

The intended use of this facility is to limit the sizes of arrays which are only partly full, and to allow Basis to access dynamic arrays whose actual length is being managed by the user rather than through the Basis routines `allot`, `change`, and `basfree`.

setlimit("name", "(dimension)") can be called from user or compiled code. The name may include a package specifier. The parentheses in the second argument are required. The restrictions on dimension are the same as for regular dimensioning strings: the contents of the string must consist of names, constants and operators which can be evaluated using name's database. The allowed operators are `+`, `-`, `*`, `/`.

70.7 Compileas Option

`compileas(dimension)` Authors may add the keyword `compileas` to the description of any dynamic variable in the variable description file.

The keyword `compileas` must be followed by a dimensioning string. This will be the dimension used to declare the variable in Fortran. The ordinary dimensioning string will be used by the Basis interpreter.

The `compileas` specification can also be given in the attribute section of the group header, in which case it applies to all dynamic variables in the group. Should any of those variables also contain a `compileas` specification, the dimensioning string on a variable applies to subsequent variables in the group.

70.8 Functions

It is possible to make the compiled functions in your program executable interactively by the user. All you do is add the name of the function and its calling sequence to your variable description file. The format is the same as for a variable, except that the "`dimension`" information becomes the calling sequence, and you add the word "`subroutine`", "`function`", or "`builtin`".

`function` or `subroutine` The initial letters of the names of the parameters listed in the calling sequence determine the type of argument expected in that position, unless the name is followed by a colon and then a type `integer`, `real`, `double`, `complex`, `logical`, `string`, or `external` (The type `string` means an input variable of type `character*(*)`; The type `external` means the argument must be the name of another compiled function which has also been declared in some variable descriptor file).

The type of the function itself determines what Basis expects as a return value, and can be `real`, `double`, `integer`, `complex`, `logical`, or `character*(n)` where `n` is an integer. Subroutines have no return value and the type, if given, is ignored. We recommend explicitly typing real arguments rather than relying on the first letter convention. This is required when using the "`Real8`" facilities in `mio` or `mppl`.

A frequent problem is that there may be a great many arguments to a function, so that the calling sequence must be described over more than one line. Simply break the definition after a comma to continue it to the next line.

Here is an example. The function gamma expects two real arguments and a complex argument, and returns a real value in grams/cc:

```
gamma(a,b,w:complex) real function [g/cc] # comment
```

At run-time, the arguments a user passes to `gamma` interactively will be checked for type and converted to the correct type if possible. Arguments are not checked for length. The user can also pass an argument by address by using an ampersand in front of the argument as in the following example:

```
call gamma(\&x, \&y, 7)
```

The function gamma will be called with arguments `x`, `y`, and 7. The arguments `x` and `y` will be called by address and therefore not checked for type. The third argument, 7, is not of the correct type so it will be changed to the complex value `(7.0,0.0)` and passed by value.

When arguments are passed by value there are no side effects unless the module being called modifies some variables in common blocks. That is, if the function modifies one of its arguments, it will be modifying a copy of the actual argument specified, which will then be thrown away. So it is important to pass an argument by address if it is an output variable.

`builtin` This keyword declares a built-in function. In this case the dimensioning string is only used for documentation. The units string is used to declare the number of arguments expected. This can be a single integer or a range such as `[1-5]`. A built-in function can handle a variable number of arguments and can return an array, while regular compiled functions or subroutines must have a fixed number of arguments. However, built-in functions have to be written using special facilities. These are described in section [Ref: wrbinfxns], "Writing Built-in Functions".

70.9 Making Arguments Optional

The parameters listed in the calling sequence are normally separated by commas. If one of the commas is replaced by a semicolon, or the parameter list begins with a semicolon, the arguments following the semicolon are optional. When the user calls the compiled function from Basis with fewer than the maximum number of arguments, the omitted optional arguments are supplied by Basis. The symbolic integer `DEFAULT` contains the value passed for numeric arguments, the default logical is `FALSE`, and the default string is a blank character. Note that you must `never` assign a value to the formal parameter representing an optional argument. Note also that this doesn't mean that you can omit arguments when calling from Fortran. Here is an example of a routine which has an optional scale factor `scale` which is to default to 1.0. The source is:

```
real function scaled(x,scalein)
     # scalein is optional input, never assign to it
real x, scalein, scalef
if(scalein == real(DEFAULT)) then
   scalef = 1.0
else
   scalef = scalein
endif
return(x*scale)
end
```

and the entry in the variable descriptor file is

```
scaled(x:real; scaleg:real) real function
     #returns x*scale, scale defaults to 1.0
```

70.10 Commenting the Variable Description File

Comments may appear anywhere in a variable description file after the first line that contains the package name. Comments that follow group or variable descriptions are available to the user of Basis at runtime.

The first # comment is used in Basis to label the variable in queries or printouts, a $ for development comments. Here are some sample variable descriptions that include comments:

```
x1(200) [cm] #zone descriptions

x2 integer (4) $this should be changed to 6!
   [pounds] #weights of contributors to this package.
   #The heavier the contributor the more weight given
   #their terms in the least squares solver?

file3        Filename     /"taradim"/
   #File containing geometry specification.
```

Here `x1` is a real array of length 200 containing information in centimeters, `x2` is an integer array of length 4 containing information in pounds, and `file3` is of type `Filename` and is given the initial value `"taradim"`. When `x1` is displayed, it will be labeled “zone descriptions”. `x2` will be labeled “weights of contributors to this package”, not “this should be changed to 6!” since the latter begins with a $.

All variable descriptions are input free-form, but each comment must be complete on one line.

Note also that

```
x(200)  [sec] #x is a nice variable /22./       WRONG!
```

will not give `x(1)` the value 22. because comments extend to the end of a line. Dimension strings and initial value specifications can be carried over more than one line by making the last character on a line a comma. This is usually only necessary for dimensioning strings in the case of functions that have long calling sequences. There is no limit (except that of prudence) on the total number of characters for any dimension string or initial value specification.

Succeeding groups have the same form: one or more asterisks (or reserved word `Group`) to warn of the start of the group, the group name, group attributes, a colon, optional comments, and then one or more variable descriptions.

Users frequently ask whether they can use FORTRAN-style `parameter` statements in groups. This is one place where the ‘%’ notation comes in handy. Putting a ‘%’ in column 1 causes mac to

ignore the statement, except to strip off the '%' sign, and pass it on to `mppl` and FORTRAN for processing. For example:

```
***** Mygroup :
%      parameter ( N = 25 )
x(N) real
```

The `parameter` statement will be passed along to `mppl` and FORTRAN. Sometimes users prefer FORTRAN `parameter` statements because of FORTRAN scoping rules. The parameter will only be known in the FORTRAN modules which '`Use`' the group. Basis-style parameters declared at the top of the variable descriptor file are known globally.

70.11 User Defined Types

A limited facility for user defined types is available. An author can declare a word to stand for a symbolic type by including a `usertype` statement anywhere after the package name but before the first group. `usertype` definitions may be interspersed with parameter definitions. The `usertype` statement has three forms:

```
usertype name
usertype name definition
usertype name -character
```

All of these forms tell `mac` that `name` is a type. If a definition is given, then it is used for declaring the variable. If no definition is given, it is assumed this definition is supplied elsewhere (such as in a file which will be included when `mppl` is run). The last form lets `mac` know that said definition makes name a type of `character*(something)`; such types must be handled specially by `mac` so it needs to know. A definition may involve another, previously defined, user type.

The usertypes `Address`, `Filename`, `Varname`, and `Filedes` are built into `mppl` and `mac`. These are used for variables which hold pointers, file names, variable names, and I/O connectors, respectively. We especially urge you to use `Filename` as the type for any variable which will hold the name of a file. Basis will make sure you get the right size for each system. `Varname` should be used for a string variable which is to hold the name of a Basis variable.

Example:

```
#a package that includes usertypes
xyz  #xyz package developed by me
usertype boolean logical  #boolean now a synonym for logical
usertype radoption character*24
{
  n=7, m=8
```

```
}
# version 1.0
**** Group1:
x integer
y(n,m) boolean
opt1   radoption
```

This facility has some minor limitations. In particular:

```
usertype xxx         # ok by itself
usertype yyy real*8
{
  define xxx real
}
```

is allowed, (although redundant, since it could be done in a single `usertype` statement), but you can only define a macro to be a type if it has been previously declared a `usertype`.

Dynamic arrays with a user-defined type like `xxx` above may be declared by using `_xxx` as their type; declaring an identifier as a usertype automatically declares the same identifier preceded by an underscore.

70.12 Architecture-dependent information

It is possible to put information in your variable descriptor file which will only be processed for certain specified architectures. For example:

```
SYSTEM SUN4 HP700 SOL
x(10,20) real(Size8)
SYSTEM XMP YMP CRAY C90 UNICOS
x(10,20) real
SYSTEM ALL
...
```

specifies that on Sun workstations running SunOS or Solaris, and on HP700 workstations, the variable `x` will be a size 8 real (i. e., double precision), while on various Crays it will be a real. Statements following the `SYSTEM ALL` will be processed on all architectures. In general, `ALL` is the default; a `SYSTEM` statement with a list of architectures causes the statements following (until the next `SYSTEM` statement, if any) to be processed only for the listed architectures. One may also have statements such as

```
SYSTEM +RS6000 -AXP
```

which adds RS6000 to the list of architectures for which the following statements are processed, and removes AXP from that list. The statement

```
SYSTEM ALL -SUN4
```

causes statements following to be processed for all architectures except Suns running the SunOS.

70.13 Interfacing with C and C++; The Fcc Utility

70.13.1 The process is automated

As previously noted, you can interface with C and C++ automatically, using the `language` directive. Let us discuss C first. If you declare a group as `language "C"`, then any variables declared in that group will be accessible from C language routines linked with your code, and by the same (lower case) names. When `mac` processes your code, it will automatically produce C header files and source files which declare an `extern struct` equivalent to the FORTRAN common block, and C variables which are initialized to point into the `struct`. For functions declared in a `language "C"` group, `mac` will produce an interface-defining file for the `Fcc` utility described later in this section. `Fcc` then automatically produces so-called "wrapper" functions; these are C functions which are the ones actually called by Basis. Their purpose is to convert parameters to ones that C will recognize (e. q., convert FORTRAN strings to C strings), and then to call the C routines. FORTRAN names of functions called from Basis should be all lower case; the corresponding C functions should be mixed case, but otherwise be the same name. (Alternatively, the user may declare an alias for the FORTRAN name, in which case the alias will be taken as is to be the name of the C function.)

`language "C++"` groups are handled similarly, except that mac creates `extern "C" structs` for variables so that names will not be mangled in the usual C++ way. And C++ functions to be called from Basis need to be declared `extern "C"` for the same reason. For more complete details, the reader is referred to the `mac` man page.

The `mac` utility supplies a second way of interfacing Basis with C++ code (but not C), which also allows the C++ code to call FORTRAN functions. This is described in the following section.

70.13.2 The -c command line option for mac

While the `language` directive applies to individual groups only, the `-c` command line option causes an interface to be created for an entire variable descriptor file. It has the additional capability of providing a way for C++ code to call FORTRAN functions. However, the interface is somewhat less convenient, and requires the use of a special library of array classes which allow C++ to access FORTRAN arrays in the same way that FORTRAN does.

C++ functions to be accessed from Basis are declared with special reserved words `csubroutine` (for `void` C++ functions) or `cfunction` (if they return a value). `mac` creates a class whose name

is the same as the package belonging to the variable descriptor file, for example "`pkg`". All C++ functions to be called from Basis must be member functions of class `pkg`. They can be called from Basis using whatever name they were given in the file; `mac` produces wrappers which call the functions in class `pkg`. Note that no transformation of variables takes place; in particular, FORTRAN strings will not be converted to C++ strings.

FORTRAN functions to be accessed from C++ are declared in the normal way. In this case, `mac` creates an interface which allows a FORTRAN function named (say) "`foo`" to be called from C++ as `pkg::foo`. Again, there is no transformation of variables between the two languages, so, for example, C++ strings will not be converted to FORTRAN strings, etc.

FORTRAN variables accessed from C++ also must be prefaced by "`pkg::`", since `mac` puts them all in a class by that name. Accessing scalar variables is straightforward, but arrays having more than two dimensions are declared as special C++ template array classes, in order to allow C++ to subscript the arrays the same way as FORTRAN (the user should recall that FORTRAN arrays are stored in column-major order, while nearly every other language, including C++, uses row-major order).

This interface is not for everybody. Users wishing to know more details should read the `mac` man page and experiment with a variable descriptor file to get a better idea of the interface.

70.13.3 How to write an input file for Fcc

`Fcc` is a Basis utility which takes as input simple prototypes of C functions and produces "wrapper" functions which, when called from FORTRAN, convert the FORTRAN parameters to what C expects, and then calls the C function accordingly. If the C function returns a value, then that value will be returned to FORTRAN. Because the `language` directive causes `mac` to create an `Fcc` input file automatically, most users will never need to use `Fcc` directly, and can bypass this section. However, power users may wish to know how to set up their own `Fcc` input files, and how to use `Fcc` to process them.

`Fcc` accepts the interface file name as a command line argument, has two command line options, `-h` and `-c`. The option `-h` causes a file `Fcc.h` to be created which is appropriate for the current machine. The option `-c` causes the file `Fcc.c` to be created, which is the file containing the runtime support routines needed by the glue routines created by `Fcc`. The interface file itself contains a series of descriptions of C routines which are to be called from Fortran. Each of these descriptions has the form:

```
[return_type] Name(argument_list) [alias ActualCName]
```

where the square brackets denote optional items.

`Name` is the name of the C-language routine. It must be a mixed-case name if the alias clause is not given. The Fortran call should be to a routine called `name`, where `name` is the lower- or upper-case version of `Name`. The C routine called will be `Name`, or `ActualCName` if an `alias` clause is present.

`argument_list` is a (possibly empty) list of type designators separated by commas. This list should be the same length as the argument list of the C routine. `return_type`, if present, is a single type designator, or the word '`subroutine`'. A type designator is one of the following: `integer`, `logical`, `real`, `real(Size4)`, `real (Size8)`, `real(Size16)`, `Address`, `character`, or `string`. This type designator is preceded by an ampersand in the argument list if the corresponding argument is to be passed by address. The two cases where this is needed are: (a) the argument is an array, or (b) the argument represents an output argument.

The type '`string`' is handled in a special way:

1. If an argument type is '`string`', the C routine receives a C string that is a null-terminated copy of the actual argument with trailing blanks deleted.
2. If an argument type is '`&string`', the C routine receives a blank, null-terminated string of the same length as the Fortran character variable used as an actual argument. On return from the C routine, the actual argument is filled with whatever resides in the string the C-routine received, less the final null, and is then blank padded if necessary to its full length.

A string argument cannot be used for both input and output.

The type '`character`' is also handled in a special way. Because some Fortrans limit the size of a string, it is sometimes necessary to use a long array of `character*1` to hold all the characters. To pass such an array to C, use '`character`' or '`&character`' as its type; the next argument after a '`character`' or '`&character`' argument must be an integer argument telling how many characters of the character array are to be used. The C wrapper routines then use the array of `character*1` the same as a string of that length, as described above.

The type '`real`' is an abbreviation for `real(Size8)`, unless the `-r4` option for `mac` has been used, in which case it is an abbreviation for `real(Size4)`. You are encouraged to spell out the desired kind qualifier and not rely on this option. (Which is why we don't mention it in the option section, we were hoping you wouldn't notice.)

70.14 Writing Your Source

70.14.1 Introduction

By using the `/name/` facility in your variable descriptor file, you can tell Basis about common blocks in Fortran routines. In this case your existing scientific routines will need no modification. Or, you can use `mppl` as described in this section. You may choose to supply initialization or version routines as described below.

The source you write does not need to contain any logic for user input, which the user will do with the Basis language. Subroutines are available that can eliminate many formatted writes. Much code that might normally be included for debugging purposes can also be omitted since the user can inquire about the value of variables at will. Most users eliminate graphics from their source and use interpreted graphics instead.

The document “MPPL Reference Manual” (manual VI) contains complete documentation for `mppl`. There is also a manual page for it. Many authors will use just one construct, the `Use` macro. Other than that, they use standard Fortran [Footnote: On Crays, either `CFT` or `CIVIC` may be used. Most `CIVIC` extensions can be used except alphanumeric labels.].

A macro is a name recognized by the `mppl` macro processor as a special word. It may or may not have arguments; if it does, they appear inside parentheses just as arguments to a subroutine or function do. Macro names may be of arbitrary length and are made up of letters and digits, with the first character a letter. The underscore (_) may be used as a letter. We adopt the convention that at least one character of a macro name will be in upper case to help identify it as a macro. Since `mppl` is case-sensitive, a macro named `Point` is not recognized if spelled `point`.

The macros discussed in this chapter are automatically defined by `mppl`. You must avoid using the names of these macros, or the names of the built-in `mppl` macros (`define`, `include`, `ifelse`,`ifdef`, `Immediate`, `Dumpdef`, `Errprint`, `Quote`, and `Evaluate`), for any other purpose. Also avoid the names used for symbolic constants (`Pi`, `OK`, `ERR`, `DONE`, `YES`, and `NO`) and symbolic types (`Filename`, `Filedes`, `Address`, and `Varname`).

70.14.2 Declaring a Group in a Subroutine

```
Use(Groupname)
```

This causes a set of declarations to be inserted which contain the information about `Groupname` from the variable descriptor file, thus making the variables in `Groupname` known to this subroutine. `Use` statements must appear within the declaration section of the subroutine. The `Use` statement may begin in column 1.

CASE COUNTS! If you spell `Use` as “`use`” it won’t work. `mppl` macros are case-sensitive.

70.14.3 Initialization Routine

You may choose to supply a routine to be called when Basis initializes a package. If you do, you inform Basis of this by including the keyword `init` in your `gluepack` input. If you do not choose to supply this routine, `gluepack` will supply a dummy one for you. The specification for the routine is:

```
subroutine pkginit
```

where `pkg` is the name of the package. Basis calls the subroutine `pkginit` when your package needs to be accessed for the first time, and never calls it again. An automatically written routine, `pkginit0`, initializes the database and then calls `pkginit`. The call to `pkginit` may be triggered by an inquiry about variables, for example, and does not necessarily mean that `pkg` will be run at this point. Some values cannot be data-loaded easily, and this routine is a good place to do such variable initializations. An example is an array that needs a large amount of default

data, or a string that needs to contain a nonprinting character such as “Bell”. For most packages, however, this routine will consist of just a return statement.

One of the routines, `pkgwake`, contains references to all your common block variables and so is a good place to visit when in your debugger.

70.14.4 Version Routine

You may choose to supply a routine to be called when Basis needs to print a version message about a package. If you do, you inform Basis of this by including the keyword `vers` in your `gluepack` input. If you do not choose to supply this routine, `gluepack` will supply a dummy one for you. The specification for the routine is:

```
subroutine pkgvers(ius)
  integer ius
  call baspline(ius,'Your version message here.')
  return
end
```

where `pkg` is the name of the package, and `ius` an integer output unit specifier. This routine should write a message to unit `ius` (which is already open) describing the package, such as the author and version number. This message will be printed on the terminal when `pkg` is initialized, and on certain output files when they are created. We recommend you do so with `baspline` as shown so that the version message will work correctly to graphics files.

CHAPTER

SEVENTYONE

Gluepack: Putting Packages Together

71.1 config Execute Line

`gluepack`'s execute line is:

```
BASIS_ROOT/bin/gluepack -i inputfilelist -o outputfilename
```

where

`BASIS_ROOT`

is the location of your Basis distribution.

The "`-i`" is optional. If the "`-o outputfilename`" is omitted, then `gluepack` will write to a file called "`pack.m`." The `inputfilelist` should be blank delimited; if you wish to load several packages, then their descriptions may be in one file or in several files.

If you neglect or forget to specify one or more input files on the command line, `gluepack` will want to read from standard input, i. e., the terminal. You may certainly type your input to `gluepack` at the terminal if you wish. In this case, use the character ^{D} (control-D) to signify an end-of-file.

The only other command line option likely to be of interest to most users is the "`-e`" option, which echoes the input to the standard output. Normally `gluepack` produces minimal output to the terminal; however, all warning and error messages will appear there.

71.2 config Input File Format

The `gluepack` input file is mostly free format. Ends of lines have no significance except that, like commas and white space, they act as delimiters between tokens and/or statements. `config` input files may contain the four kinds of statements: package statements, array assignment statements, scalar assignments, and system specifications. Each type of statement will be discussed in more detail below.

Let us first examine the components of `config` statements, which are called *tokens*. Tokens include reserved words (the ones discussed earlier and others which will be described later), identifiers, unsigned integers, arbitrary strings (which may be enclosed in either single or double quote marks), parentheses '`(`', '`)`', and brackets '`[`', '`]`', which are used to enclose lists of items in array assignments, and the assignment operator '`=`'. Tokens may be separated from one another by white space (blanks, tabs, end-of-line), commas, or comments. A comment begins with an octothorpe '#' outside of quotes, and extends to the end of the line on which it occurs.

Enclosing a string in single or double quote marks has the effect of removing any special significance that the string or any character in it may have. Thus, if you want a string to contain spaces, commas, or a '#' character, then enclose it in quotes. As another example, `codefile` is a reserved word, while `"codefile"` is an identifier and not the reserved word, as is `'codefile'`. Single and double quotes are equivalent except that a string enclosed by one kind of quotes may not contain the same kind of quote within it. If you inadvertently omit the closing quote from a string, `gluepack` will print a warning but will accept all characters up to the end of the line on which the string began.

71.2.1 Package Statement

The package statement must be used to give a name to each package which you wish to include, and may optionally be used to specify the maximum number of calls to your `pkgexe` routine in the "`step`" phase of the `RUN` command, and to specify which (if any) of the eight standard routines you are supplying. The package statement must begin with one of the reserved words `package` or `foreign`, [Footnote: Foreign packages are described in section [Ref: foreign-pkgs] of this manual.] but otherwise its form is not unduly restrictive. It is made up of substatements whose order is quite arbitrary.

The only required part of a package statement is the substatement which gives a name to the package, which has the form

```
pkg = <string>
```

where `pkg` represents any identifier with three or fewer characters, and is called the *short name* of the package. The `string`> is any `gluepack` string, usually quoted, representing the title of the package. An example is

```
dap = "Designer's Apprentice"
```

(Note that since the title contains both a space and a single quote, it *must* be enclosed in double quotes.) The short name of the package may not be any of the two or three letter reserved words `gen`, `exe`, `fin`, `yes`, or `no`, unless it also is enclosed in quotes, but we really hope that you don't do this.

Optionally one may specify the maximum number of calls to `pkgexe` (or its substitute routine, if you specified a different name) in the "`step`" phase of the `RUN` command. This is done by means of a substatement of the form

```
limit = unsigned decimal integer>
```

If a `limit` substatement does not occur, then the default value of 10000 will be used.

Finally, you give the root names of the routines which you will be supplying for this package . These substatements take one of the two forms

```
root
```

or

```
root = <name>
```

`root` representing one of the eight reserved words `vers`, `init`, `gen`, `genp`, `exe`, `exep`, `fin`, and `finp`, and `name`> being the legal FORTRAN name of an integer function. In the former case, you must supply a FORTRAN function named `pkgroot`, where `pkg` is the name of the package; in the latter case, your function is named `name`> instead of the default `pkgroot`. Thus, for example, the substatements

```
gen, exe, fin = alldone
```

mean that you are supplying routines `pkggen`, `pkgexe`, and `alldone` (in lieu of `pkgfin`).

The substatements of a package statement need not occur in any particular order. Here is an example of a correct package statement:

```
package limits = 1000 vers , init , exe
rho = "Density Calculation" , exep = plotexe
finp = plotfin # note that quotes are not required here.
```

71.2.2 Scalar Assignment Statements

The form of a scalar assignment is:

```
variable = <string>
```

where `variable` is one of the reserved words `codename`, `cprompt`, `probname`, `verbose`, `echo`, `libpaths`, or `libs`, described in an earlier section, and `<string`> is, in some cases, restricted as noted there.

Assignments to any of these variables can be omitted; they then take on the default values noted in the table. If more than one entry is encountered for a specific variable, then only the first specification is used. A warning is issued for subsequent assignments to the same variable if a different value is specified.

71.2.3 Array Assignment Statements

The array assignment statement may take one of three forms, first

```
variable = <string>
```

if only one string is being assigned, or second

```
variable = ( <list of strings> )
```

or third (and equivalently)

```
variable = [ <list of strings> ]
```

where <list of strings> is delimited by white space or commas.

The meanings of the array variables codefile, paths, macfile, startups, firstpkg, iotable, and ncodefil have already been discussed. Multiple assignments may be made to the array variables; the effect is to add the subsequent values to the end of a list of the values assigned.

71.2.4 System Differencing Statements

```
SYSTEM <CPUlist>
```

This is the reserved word SYSTEM (which must begin in column 1) followed by a list of one or more CPU specifiers separated by white space or commas (no parentheses). Currently, the allowed CPU specifiers are CS2, SOL, SUN4, HP700, RS6000, SGI, GENERIC, XMP, YMP, C90, CRAY2, ULTRIX, VAX, MAC, and MIPS. These specifiers control the setting of toggles in gluepack, which initially are all toggled on. The effect of a <CPUlist> is to turn off all toggles except those for CPU's contained in the list, which will be turned on. Then any statements following the <CPUlist> will only be processed by gluepack if gluepack is processing for one of the CPU's in the list. gluepack is normally processing for the CPU on which it is executing, but it can be set for a different CPU by the -CPU option described earlier. The main use of this statement (and the following) is to specify the names of libraries, library paths, codefiles, object files, etc., which may differ from one platform to another. Example:

```
SYSTEM HP700
libpaths="-Wl,-L/usr/lib -Wl,-L/usr/lib/pa1.1 -Wl,-L/lib/pa1.1"
libs="+DA1.1 -lf -lm -lisamstub"
SYSTEM SOL
libpaths="-L/usr/lib -L/opt/lib -L/opt/SUNWspro/SC3.0/lib"
libs="-lF77 -lM77 -lm"
SYSTEM SUN4
libpaths="-L/usr/lang/SC1.0"
libs="-lF77 -lm"
```

```
SYSTEM +CPU or -CPU
```

Here, `CPU` is one of the allowed `CPU` specifiers enumerated above. The effect of `+CPU` is to turn on the toggle for just that one `CPU`, and of `-CPU` is to turn it off. No other toggles are affected. Examples:

```
SYSTEM +YMP
SYSTEM -SOL
```

71.3 Configuring the Packages with .pack files

Assume your package name is `pkg`. You need to create a `pkg.pack` input file as described in this section. The `pkg.pack` input file contains names and various other information about your packages. This information is of the following sorts:

- Specifying a package to be included in the program.
- Informing Basis of the presence of one or more of the eight optional routines that can be supplied for each package. Every package may have the routines `pkginit` or `pkgvers`. If you are including package `ctl`, you may also have chosen to supply one or more of the routines `pkggen`, `pkggenp`, `pkgexe`, `pkgexep`, `pkgfin`, and `pkgfinp`. The ones which are present must have their root names specified in the `pkg.pack` input file. The user may also supply alternate names for these routines.
- Customizing the appearance and behavior of your program. `pkg.pack` can set various "customizing" variables which tell the system what you want to use as a prompt, for instance..

Given this information, the `gluepack` utility writes a series of routines required by Basis, generates the calls to the routines which you are supplying for each package, and sets the customizing variables.

71.3.1 Sample .pack Input File

Here is a typical `.pack` input file.

```
package tri = "Trivalent Unit Flow Descriptor" init
firstpkg=tri
codename = "Trivalent"
cprompt = "Tri> "
echo = no
```

This indicates that one package, named `tri`, is to be loaded, and that the package `tri` has an initialization routine `triinit` that is to be called when `tri` is initialized. The `firstpkg=tri` tells Basis to initialize `tri` when the program starts up. The next three lines customize the program name, its prompt, and cause it not to echo input read from files to the terminal.

71.3.2 Short Tutorial on the gluepack Input File

`gluepack` input files may contain two kinds of statements: package statements (which begin with one of the reserved words `package` or `foreign`), array assignment statements (which begin with one of the reserved words `codefile` (formerly `macfile`), or `firstpkg`), and scalar assignments (which begin with one of the reserved words `codename`, `cprompt`, `probname`, `verbose`, `echo,`).

Assignments are the easiest to understand, because they always take one of the three forms

```
variable = <string>
```

if only one item is being assigned, or second

```
variable = ( <list of strings> )
```

or third (and equivalently)

```
variable = [ <list of strings> ]
```

The variables which can be assigned single values (scalar variables) are:

codename The code name (1–8 characters; default: Basis).

cprompt The prompt to use (1–16 characters; default, `Basis>`). If the prompt contains spaces or other characters with special meaning, it must be enclosed in quotes, thus: `"Basis> "`.

probname `probname` sets the Basis variable `probname` on startup. This is a deprecated feature that will be removed in the future.

verbose The initial value of the parser variable `verbose`. Specify `yes` or `no`. Many of the system messages to the tty (and logfile) will be eliminated if `verbose = no`. (default: `yes`)

echo The initial value of the parser variable `echo`. Specify `yes` or `no`. This controls whether or not input files are echoed to the terminal when they are read. (default: `yes`)

The variables which may be assigned either single values, or lists of values enclosed in parentheses or brackets and delimited by white space or commas are:

codefile `codefile` is a list of search directories for Unix. Whenever Basis tries to open a file and cannot, it then will try to search each directory in this list. This list will be searched in the reverse of the order it is specified, and prior to the default search path. See `paths` (below) for the opposite search order. If you install your program somewhere, use `codefile` to let Basis find your comment files, standard input files, etc. The list can be separated by either blanks or commas. The strings assigned must be legal file names on whatever system you are using. See the routine `pathadd` in the Basis Language Reference Manual for details about the default search path.

firstpkg The initial Basis Language search stack. The top of the stack should be on the left (or first if there are several `firstpkg` specifications). Each package is initialized as it is placed on the stack.

Strings assigned must be legal package names (identifiers of length 3 or less). Normally, every package should be mentioned in a `firstpkg` statement, since typically you will want each package initialized. (default: parser only).

iotable If you have a code that you wish to convert to Basis you may wish to reserve one or more I-O unit numbers so that the rest of Basis will not use them. To reserve units 1, 2, and 61, enter: iotable = (1,2,61)

WARNING: units 5, 6, and 59 can't presently be reserved.

path A list of directories for unix. Similar to codefile, except that this list of directories will be searched in the order in which the directories are listed, but still prior to the default search path. **startup**] The name(s) of (an) input file(s) for the program to read before it begins reading any user input. These files will be read in the reverse of the order listed. A program which reads such a file can thus read in a custom set of user-defined functions or a set of custom parameter settings. The files should be somewhere where the code can find them, see `path` above. The list can be separated by blanks or commas. Strings assigned must be legal file names on whatever system you are using. (default: none).

If you wish to end the run immediately after executing the `startup` files, set `notty = yes` in a macfile.

A startup file will be treated specially in the following two cases: a. If the first character is a period, Basis will silently continue if it cannot find the file. b. If the first character is a dollar sign, Basis will substitute the value of the environment variable whose name follows.

All Basis codes have .basis and $BASIS as startup files. .basis is read first, followed by $BASIS, if set, followed by any code-specific startup files.

71.4 config Errors

`gluepack` has three levels of errors, given below in increasing levels of severity. Each type of error causes an appropriate comment to be sent to standard output, and additional actions as described below.

- Warnings. These include attempted reassignment of a scalar variable (the first value assigned will be retained), a string with no closing quote (the rest of the current line will be taken), and renaming a package (the most recent name given will be taken). After a warning, processing continues as if nothing happened. The file `pack.m` will be written if only warnings occur during processing.

- Syntax and semantic errors. When these errors occur, scanning of the current `gluepack` statement is terminated, and `gluepack` proceeds to the start of the next statement. The writing of the output file `pack.m` will be suppressed. There are many such errors, e. g., attempting to give a package a name of longer than three characters, assigning something other than `yes` or `no` to `echo` or `verbose`, attempting to match a '`(`' with a '`]`', and the like.

- Fatal errors. These will cause instant termination of execution. They are incorrect command line, inability to open an input file, and the occurrence of a nonprintable character [Footnote: However, if reading from the terminal, `^{D}` (control-D) will be accepted as an end-of-file.] in the input.

CHAPTER

SEVENTYTWO

Programming Support Facilities

72.1 Specifying Variables' Names

Many of the routines in Basis can access variables by name. They do this by searching a run-time database that is available for each package. It is important to be sure that the name given specifies the desired variable completely. If there should be a variable of the same name in another package, confusion may result. Oh, Basis won't be confused, but you might be. Basis maintains a stack of open packages and will find an unqualified name in the highest package in the stack in which it occurs, which may well not be what you want.

The name of a variable can be prefixed with the name of the package and a period, as in:

```
call edit(STDOUT, "pos.x")
```

which writes the value of variable `x` in package `pos` to the terminal. If you are sure that the name of a group or variable is unambiguous, and that the package in which it resides is sure to be on the search stack at the time the call is made, you may omit the package prefix. A prefix consisting of '`local`' as in

```
if( exists("local.x") ) then...
```

restricts the search to the local variables of the current user-defined functions, while a prefix consisting of global as in `"global.x"` restricts the search to the user-defined variables.

One time in which the package in which a variable resides will *not* be on the search stack is during execution of theinitialization routine `pkginit`. If you wish to call `edit`, `allot`, etc., from this routine, you *must* give the package prefix as part of the name.

72.2 Dynamic Dimensioning

Basis allows the use of variables that change their size depending on the size of the problem. To make a variable of this type, called a dynamic variable, precede the type of the variable with an

underscore in the variable description file, give it a dimension that is a function of variables which contain the size desired, and then, once your code is running, call `allot` or `gallot` (described below) after the size is known but before the variable is used. This section explains the use of dynamic dimensioning in detail.

72.2.1 Declaring Dynamic Variables

Normally a variable entered in the description file is made visible to a subroutine when the statement

```
Use(Groupname)
```

is encountered in the declaration section of the subroutine. The `Use` statement is expanded by the preprocessing pass to statements

```
type var
dimension var( dimension ) #if specified
common / pngm / var
```

for each name `var` in `Groupname` and its corresponding type and dimension information. Here `pngm` is a name unique to this package, group, and type (unless the user specified a name in the group header).

To declare a dynamic variable (a variable whose location is determined by the contents of another variable, called its pointer) use an underscore as the first letter of the type, e.g.,

```
var(n) _real
```

This generates:

```
type var
integer Point(var)
pointer (Point(var), var )
dimension  var(n)
common /pngm/ Point(var)
```

If you are on the Sun or are running `mppl` with the `-DCOMPILER=CFT77` option, the `integer` statement is removed. On 64-bit machines the `integer` statement becomes `integer*8`.

Here, `Point(var)` is a macro that expands into the name of the pointer by prefixing `var` with the letter '`p`'. Dynamic variable names should be chosen to have at most seven characters so that `Point(var)` will be a unique identifier. If the letter '`p`' is not a good choice for your code you may change it by including a statement like

```
%define([Point],Z$1)   #change pointer initial to Z
```

in your parameter section. The percent sign causes this line to be put verbatim in the `mac` output file `macpkg`.

Each dimension can include any integer expression involving constants and names of variables. For example,

```
n     integer
m     integer
xx(n, (m + 1)/2)   _real
```

creates a variable `xx` whose dimensions depend on variables `n` and `m`. After `n` and `m` have been set, a call to `allot` such as

```
call allot("pkg.xx",0)
```

will compute the value of `n * (m + 1)/2` and then allocate that many elements of storage for `xx`. In resolving such variable names, the package to which the variable belongs will be searched first, followed by the normal search stack.

The `allot` subroutine is described in detail below.

72.2.2 Run-Time Routines

The following subroutines are used for a dynamic array that is visible to the run-time database manager.They may be called from the Basis command line at runtime, or they may be called from Fortran code. The lengths used in the subroutines are element counts that are independent of variable type.

Each of the following six routines is actually an integer function. They return a value of 0 if they executed correctly.

allot call allot(”array”,length)allocates a variable named `array` of `length` elements. The elements are initialized to 0 (or blank for character types), or to a value specified in the variable description file. The quotes around the array name are required. If `array` is a multidimensional array, `length` is the length of the desired last dimension of `array`. The database manager calculates the type and other dimensions of `array`. If `length` is negative or 0, the database manager also calculates the last dimension. Each element would contain 2 words if `array` is complex, for example.If the array has already been allocated space, the old space is released before reallocating and no error occurs. If you wish to check the value returned by `allot` you would do something like:

```
integer allot
external allot
...
if( allot("array",length) .ne. 0) then
     ....error handler goes here
endif
```

The parser variable `padding`, whose default value is 0, can be set to a positive integer by the user. This value is used as a number of elements to be added to the end of the space allocated by `allot`. This space is initialized by `allot` but thereafter is not used by Basis in any way.If the argument `length` is negative, its absolute value is added to `padding` to determine the amount of padding for this variable.Similar remarks apply to `basfree`, `change`, `gallot`, `gfree`, and `gchange`.

basfree call basfree(”array”)releases space for `array` previously obtained by a call to `allot`. See `allot`.

change call change(”array”,newlength)changes the length of `array` to `newlength`. `change` is otherwise the same as `allot`, except that it preserves the previous contents of the array. The new elements are initialized to 0 (or blank for character types), or to a value specified in the variable description file. If you call `change` with the name of an array that has not yet been allotted; `change` will call `allot` for you.If an array is multiply dimensioned and some of the sizes of the dimensions change, the old data is correctly selected and repacked in the new space. If no sizes have changed, the array is not moved. The algorithm used in its full generality is given below. Simply stated, if a dimension shrinks, the contents get deleted, and if it expands, new space is added.If the current size of name is `old(i), i = 1, nold` and the desired new size of name is `new(i), i=1,ndim`, then

1. The new size is `new(i), i=1, ndim`
2. This space contains the data from the subobject of the original object described by:min(old(i),new (i)), i=1,min(nold,ndim) 1 , i=min(nold,ndim)+1, nold
3. This data is stored in the subobject of the new space described by:min(old(i),new(i)), i=1,min(nold,ndim) 1 , i=min(nold,ndim)+1, ndim
4. The new object has its lower/upper indices derived from the current evaluation of its dimensioning string. Any limiting string is ignored by `change`.
5. If the new and old sizes agree, the array is not copied to a new location; `change` has no effect.
6. As before, if the second argument is greater than 0, the value is used to replace the value of `new(n)` calculated from the dimensioning string.

If the second argument is less than 0, then `new(n)` is not affected. A padding of `-n` elements is added to the end of the storage for the array. Basis promptly forgets about this padding. This padding is in addition to the value in the Control variable `padding`.This routine must not be called if the array has been allocated space by the author using `osallot` rather than

`allot`, unless the author subsequently calls the routine `setshape` so that Basis is aware of the current size of the array. See `allot`.

gallot call gallot(”Name”,n)calls `allot` for all the dynamic arrays in the group, `Name`. See `allot`.

gchange call gchange(”Name”,n)changes the allocation of all the dynamic arrays in the group, `Name`. See `change`.

gfree call gfree(”Name”)frees all the dynamic arrays in the group, `Name`. See `free`.

72.2.3 Using the System Memory Manager

If you wish to allocate space dynamically from within a Fortran routine without using the above facilities, you can do so by using the Dynamic and Point macros described in the next section and then calling the following routines:

osallot call osallot(ipointer, length)allocates an array of `length` words. You are calculating this number. The first argument is a variable which is returned containing the address of the allocated space. It should have been declared type `Address`. If `osallot` cannot allocate the desired space it returns to the user via the routine `kaboom`.

osfree call osfree(ipointer)releases space located at the address in `ipointer`, which should have been declared type Address. If `ipointer` does not contain a correct heap manager address control returns to the user via `kaboom`.

oschange call oschange(ipointer,newlength,oldlength)changes the length of the space pointed to by `ipointer`,which should have been declared type Address. Again, `kaboom` is called if anything goes wrong.The variable `ipointer` in all these examples either has been declared of type `Address` (an `mppl` macro which expands to the correct type on all architectures), or has been declared to be the pointer to some dynamic variable. An easy way of creating such variables is given in the next section.

72.2.4 Dynamic Array Macros

You can usually avoid the use of the following macros. Declare the variables as dynamic in the variable description file and `allot` and `basfree` them as necessary. Or, declare them in a local group and `Use` it in a subroutine. The declarations required will then be taken care of by the preprocessing system.

Dynamic

```
Dynamic(array,type,dimstr)
```

Creates a local dynamic array that is not visible to the run-time database manager. It declares array to be a local, pointered variable of type `type`, and dimension `dimstr`. If `dimstr` is omitted, `array` is declared to be a scalar. For example,

```
Dynamic(iout,integer,1)
```

declares `iout` to be an integer one-dimensional array, while

```
Dynamic(j2d,real,[5,1])
```

declares `j2d` as a two dimensional real array dimensioned `(5,1)`. Note the use of square brackets to protect the comma in the third argument from `mppl`. `Point(array)` must be set to some location (usually by `allot`, `osallot`, or assignment from the `loc` of something) before the variable is used.

Point

```
Point(var)
```

The `Point` macro returns the name of the pointer to `var`, which must have been previously declared `Dynamic`. Use this macro if reference to a pointer is needed. For example, if `acol` is a pointered variable declared by `Dynamic(acol,real,1)`, then

```
call osallot(Point(acol),100)
```

allocates 100 words of storage for `acol`.

72.3 Output Routines

Basis provides facilities for sending messages to the terminal, creating output files, writing edits of variables, etc. One constraint on authors is that you can't simply pick a unit number, open a file, and start writing to it. Instead, you must use a variable to hold the unit number and use `outfile` or `absfile` to create the file and return a unit number for you to use. This procedure allows different packages to operate independently without conflict.

72.3.1 Writing Messages to the Terminal

remark and Other Choices

The preferred way to do output to the terminal from a Fortran routine is:

```
character*80 msg
......
write(msg, format) ...
call remark(msg)
```

This example assumes that the format only writes one line of 80 characters or less. To write multiple lines with one format, make `msg` an array, write to the array using a multi-line format, and then after the `write,` loop over the call to `remark`.

```
call remark( string )
```

causes `string` to be displayed at the terminal. `string` may be a constant character string, the name of a character variable, or even a character expression. `remark` may be called from Fortran or at runtime from the Basis command line. `remark` folds long lines and uses `baspline` and `iooutus`.

`iooutus()` is a function that returns to a FORTRAN program the unit number of the current Basis output. The Basis command `output` can be used to redirect terminal output to a file. Using `iooutus()` as a unit number conforms to the `output` command.

```
write(iooutus(), format ) ...
```

By contrast, `STDOUT` is a symbolic constant representing the unit number of the controller. `STDOUT` is defined for you by `mppl`.

baspline, baswline

`baspline` is called from a Fortran routine:

```
call baspline(iunit,msg)
```

where `iunit` is a unit number (or `STDPLOT`) and `msg` is a character variable containing the desired message. Another routine, `baswline`, is called in the same way. `baswline` calls `baspline` and then calls `ruthere` to check for interrupts. Use `baswline` instead of `baspline` if you are willing to have the program return to the prompt.

baspecho

The routine `baspecho` is used to create a kind of log file. It may be called from Fortran code or from the Basis command line.

```
call baspecho(iunit)
```

`iunit` should be the unit number of a currently open file, or with `iunit` = `STDPLOT` for output to the graphics package. (Call with `iunit` = 0 to disable.) The internal variable `iecho` is set to `iunit`. Then:

1. Subsequent calls from Fortran code to `baspline` or `baswline` with a unit number of `STDOUT`, `STDERR`, or `STDPLOT`, but not equal to `iecho`, will echo to this unit number. If `iunit` is not open on a file, then Basis disables the echo, issues a warning message, and calls `kaboom(0)`.
2. Input lines read from the terminal will also be echoed, preceded by the characters '> '.

Since most Basis output to the terminal is via `baspline`, such a file will be a close approximation of a log. From the parser one could open such a file with either `outfile` or `basopen`. Such a unit opened with `basopen` but then passed to `baspecho` will lose its property of being closed when errors occur.

Example: make an almost-log file:

```
call baspecho(basopen("Log","w"))
```

Example: graphics log

```
call baspecho(stdplot)
```

Any given application program may, of course, be writing directly to the terminal using `write` statements, without going through `baspline` or `baswline`. Such writes cannot be caught by `baspecho`.

baderr

```
call baderr(string)
```

This can be called from Fortran only. `baderr` is the same as `remark` except that it terminates the program after issuing the message. The name of the calling package or routine should be used as part of the message. This routine should only be used for errors which indicate irreparable damage has occurred and no further problems can be run. For a softer escape see `kaboom`.

72.3.2 Creating Output Files

outfile

```
call outfile(myout,"comment")
```

creates an output file for the package. Subroutine `outfile` fills the variable `myout` with an integer value, the unit specifier for the file so created. `Myout` should be used as the unit specifier in all formatted write statements to the output file. Multiple output files may be created by one package. The comment (which must be enclosed in quotes) will be displayed when the program terminates, along with the name of the output file. If calling from the Basis language rather than from Fortran, be sure to pass `myout` by address (`outfile(&myout,...)`).

basopen

```
integer basopen
iunit = basopen(name, access)
```

This routine is used for opening input files and for creating output files. It may be called from Fortran or from Basis. If called from Fortran, and opened with access `"w"`, `iunit` may be used in subsequent calls to `baspline` or `baswline`, and also, of course, in Fortran `write` statements. If called from Basis, `iunit` may be used as the target for stream output.

If access is `"r"`, `basopen` opens file `name`, returning the unit number to use in subsequent operations. If the file is not present, it is searched for (using the list in variable `path`, which can be added to with the variable `codefile` in `gluepack`, or by the routine `pathadd`, described below). Error recovery is invoked if the file cannot be found at all.

If access is `"i"`, `basopen` returns `OK` or `ERR` (0 or -1) to indicate whether or not the file can be opened in `"r"` mode.

If access is `"w"`, the file is created in the current working directory, returning the unit number to use in subsequent operations. Error recovery is invoked if the file cannot be created.

Any file opened with `basopen` will be CLOSED whenever error recovery takes place. Files created with `outfile, however`, are NOT closed when an error occurs.

basclose

```
call basclose(myout)
```

closes a file that has been opened in any manner. `basclose` is accessible from both FORTRAN and Basis.Files will be closed when the program terminates if they have not been closed already.

freeus

```
call freeus(myout)
```

sets `myout` to a free unit number. You must immediately open a file on it to preserve your reservation. Use of `outfile` is preferable. This routine may only be called from Fortran.

pathadd

An alternative to specifying directories using `gluepack`'s codefile specifier is to call `pathadd` with the name of the directory. `pathadd` may be called from either Fortran or Basis.

```
call pathadd(directory)
```

The only difference is that paths added in this way are not available for search at the very beginning of the program when searching for start-up files.

72.3.3 Printing Variables and their Attributes

edit

```
call edit( myout,"name" )
```

prints the contents of the group or variable whose name is `name` (the quotes are required). The output is written on the file connected to `myout`. Example:

```
integer myout, basopen
myout = basopen("myfile","w")
call edit(myout,"pr.Geometry")
call basclose(myout)
```

would write the contents of all variables in the group named `Geometry` in package `pr` to a file `myfile`. This code will work both in Fortran and in Basis. If attempts to find the desired name fail, a remark to that effect is written instead.

list

```
call list(myout,"name")
```

is the same as `edit` except the output consists of a description of the variables and their attributes instead of their contents. It may only be called from Fortran. (Basis has a '`list`' command which may be used instead.)

72.3.4 Plotting

The `EZN` Graphics Package is the standard graphics package available with Basis. It uses the `NCAR` Graphics Package. A separate manual (III) is available to describe the plotting package. For authors who wish to supply a different graphics package, Basis expects there to be a routine

```
call ptext(msg)
```

which is to write messages on graphics frames, if desired. The user-supplied `ptext` is responsible for frame advances, etc.

72.4 Replaceable Routines

There are some routines which you can replace with your own versions. You merely need to be sure that the binary for your routine is encountered first in the load process.

72.4.1 User main routine

Basis calls a subroutine `usrmain` immediately after collecting command line arguments. If you need to do special initialization or to process the command line yourself, provide your own version of `usrmain`. Normal basis error recovery procedures are not yet installed at this point. The default `usrmain` calls `basmain`; your replacement needs to do that too. Any remaining text in `cmdline` is treated as the first line of input by `basmain`.

```
subroutine usrmain(argv0, cmdline)
character*(*) argv0,cmdline
call basmain(argv0,cmdline)
return
end
```

72.4.2 Custom handling of input

Each line read from an input file is made available to a user-replaceable routine called `basisech`. The default version (see below) does nothing.

```
subroutine basisech(line,nline)
character*(*) line
integer nline
return
end
```

72.4.3 Error handling

When Basis encounters an error in its input, it normally calls a routine named `kaboom`, to re-initialize the parser and restore data structures to a clean state if possible. During error recovery, it calls a user-replaceable routine named `basiserr`, which, by default, does nothing:

```
subroutine basiserr
return
end
```

72.4.4 Signal handling

It may be useful for your code to catch certain Unix signals and do special things. For example, some batch job systems use SIGTERM to tell a process to exit gracefully. Codes running under such a system might catch SIGTERM and make a restart file before exiting. The default routines, as shown below, call internal handlers that result in your code exiting immediately after receipt of any of the signals TERM, URG, USR1, USR2.

```
subroutine basterm
call dosigterm
end

subroutine basurg
call dosigurg
end

subroutine basusr1
call dosigusr1
end

subroutine basusr2
call dosigusr2
end
```

72.4.5 Code load time and date

As Basis starts up, it prints various information to the terminal or other output logs. Among this information, it is often useful to record the time and date at which the particular code you are running was built. The routine `glbtmdat` is intended for this purpose. The default version, shown below, enters blanks for your code's load time and date. The typical approach for replacing this routine is to construct and compile it automatically as part of your Makefile dependency tree for the code itself.

```
subroutine glbtmdat(codetime,codedate)
character*(*) codetime, codedate
codetime = ' '
codedate = ' '
return
end
```

72.4.6 Conversion Considerations

Here are some of the things to watch out for when converting existing code to Basis.

- A source of possible problems that are easy to fix, but are often difficult to find occurs if the user's source has modules with the same names as routines in the Basis system. The Unix `nm` utility can help create lists of names, and of course loader output must be scrutinized.
- Unit numbers used for output files must be reserved using the `iotable` feature of `gluepack`. Alternatively, use `freeus`, `basopen`, or `outfile`.
- Let Basis do as much as possible. Many calculations and plots can be done with the interpreter, reducing the amount of Fortran you must maintain. One of the surprising developments as people got used to Basis was the migration of tasks that used to be in Fortran up into the interpreter. You can use a startup file (see `macfile` in the `gluepack` documentation) to read in interpreted Basis Language code as your program starts.

72.5 Symbolic Constants

The following symbolic constants are defined by `mppl`:

`DONE` A symbolic integer indicating completion of an iterative process.

`ERR` A symbolic integer indicating an error.

`NO` A symbolic integer different from YES; used to indicate a negative condition. Actual value is 0.

`OK` A symbolic integer indicating success.

`Pi` pi =3.14159...

`YES` A symbolic integer different from NO; used most commonly to test conditions. Actual value is 1.

72.6 Symbolic Types

Symbolic types are used just like ordinary Fortran types such as `integer` or `real`. They are changed by the macro processor into suitable definitions for the target machine. Their use makes it easier to read, understand, modify, and port code. The currently defined symbolic types are:

`Filename` a character variable big enough to hold a legal filename. Usually about 256 characters.

`Filedes` integer variable that holds an i/o connector number.

`Varname` character variable big enough to hold a Basis variable name.

`Address` an integer long enough to hold a pointer. On most architectures, this is the same as a Fortran integer, but on 64-bit architectures it is an `integer*8`.

72.7 Physics Unit Codes

Unit codes are text strings containing the units of physical data. Currently they are only used to label output and to improve the documentation of the variables. The following unit codes are suggested.

`m` Meters

`s` Seconds

`g` Grams

`V` Volts

`A` Amperes

`eV` Electron volts

`rad` radians

`None` Ordinal or dimensionless quantity

The units above may be modified and combined. The modifiers are:

`u` 10-6

`m` 10-3

`c` 10-2

`d` 10-1

`k` 103

`M` 106

`G` 109

`T` 1012

To combine units use `* , /, **`, and parentheses. For example, we have:

```
cm/s**2          [centimeters per second per second]
V*A/cm**2        [volt-amperes per square centimeter]
```

72.8 Interfacing with C and C++ Programs

See the chapter “Writing Basis Packages” for details.

72.9 Communication Between Packages

72.9.1 An Editorial

The big problem in large code development is how to prevent the program from getting harder and harder to change until finally no one is willing to work on it. I call the resistance of a code to change its “inertia”, and a goal of the Basis System is to minimize inertia. In my experience, the main contributor to inertia is the methods used to communicate between different pieces of physics (especially where there are multiple authors).

Consider two packages A and B, where A needs to know some quantity `rho` calculated by B. There are many ways in which A could get `rho` from B, but the most frequently used method is for both A and B to declare some common block containing `rho`. Most typically this is done by means of a macro statement which declares an entire common block.

Consider the consequences: the author of B now has to watch out that she doesn’t use any of the other variable names in the cliche, even though she may have no use for these variables. If `rho` is not the name she prefers for that quantity she may be tempted to alias something to `rho`, thus leading casual scanners of her source to believe that she doesn’t use `rho` at all. If A wants to add more variables to his cliche that declares `rho`, disaster may strike B. Worse, B has to be recompiled in order to change A.

Now suppose that `rho` represents a spatial quantity `rho(x)`. Suppose that A has represented `rho` by having a grid `x(j)` and values `rho(j)` that correspond to `x(j)`. B now needs both `x` and `rho`. If B needs values of `rho` at values of `x` not represented in the grid, she needs to use a table lookup and interpolation scheme. Perhaps A does too, thus leading to duplication of code, or worse, a different interpolation scheme being used in each package leading to an inconsistency in the representation of `rho` in the program as a whole. Then comes the day when A learns of a dramatic new breakthrough in calculating rho that involves using a finite-element representation. But to install it, A must track down every other package that uses `rho` and change how THEY access `rho`, too. The inertia of the program may discourage this improvement.

To my mind, the source of the problem is that B, a consumer of `rho`, has no business at all knowing how `rho` is produced. It is far better if A supplies a function `rho (x)` that returns the value he has produced. If there are some parameters in the production of `rho` that might need to be set by another package, A can write a function for B to call that sets the parameter.

There may be a few places in a large program where this leads to efficiency problems; in those places one could get the information by calling a function that returned appropriate pointers. But the need should be strong before resorting to sharing a representation in that way, and an interpolation function should be provided so that the quantity is consistently treated.

This editorial was written in 1984 and is left here for historical reasons. Now that we all do object-oriented programming, you all believe it already, right?

72.9.2 Global Common

If you wish to set up a global common, create a package containing the groups you wish to be known to all other packages. Typically this package would have little or no source, perhaps only the `pkginit` routine. Or, it might be the "driver" for the other packages. If packages residing in other directories need access to these variables, they should list the path to this variable descriptor file in the `NVDF` category of their `Package` file. This causes the "global" variable descriptor file to be processed first and its definitions made available in preprocessing the source.

72.10 The Package Library

Packages can be shared. If you develop a package which might be useful to others as a component of their programs, please let us know about it. The chapter "Basis Package Library" describes packages available to you.

CHAPTER

SEVENTYTHREE

Advanced Package Writing

73.1 There Be Dragons Here

The purpose of this section is to warn you to stop reading this chapter NOW. The following sections are of interest only to a small minority of those who will use Basis. Before you decide that you need to use any of the following facilities, you might contact us and describe your problem. We often know an easier solution.

This chapter covers accessing interpreter variables from compiled routines, writing "foreign" packages which have variables not declared in the usual way, and writing your own built-in functions and attribute handlers.

73.2 Accessing Variables from Compiled Routines

Sometimes you may need to access a variable owned by another package or declared interactively by the user. The following routines are used to access a variable by name. You have the choice of specifying which package to search or of searching the current stack. The basic procedure is to use routine `parfind` to find the variable and its type, and then routine `rtxdb` to get the location and size of the variable. The types returned are integer codes with the values such as `NULL` = 0, `INTEGER` = 1, `REAL` = 2, etc. A value of less than zero indicates a character variable holding that many characters, e.g., the type code for `character*8` is -8. Other values indicate items like functions and structures. The proper way to interpret these codes is by using the functions `utcodstr` and `utstrcod` as explained below in section [Ref: wrbinfxns] , on writing your own built-in functions.

73.2.1 Finding a Variable

There are two routines available for finding a variable in the database. The first, `parfind`, is used when you have a separate name and package number. The second, `rtfinder`, can be used on names which may contain a name of the form `pkg.name`, `.name`, or `..name`. This routine issues a message if the variable does not exist.

Function `parfind` looks for a variable given a name and package number. See routine `glbpknum` below for converting a package name to a number.

```
function parfind(npack,name,jvar,ndb,tc)
# input:
#    npack    package number of package to search,
#         or zero to search current stack
#         -1 means ONLY search local variables of
#            latest user function
#         -2 means ONLY search global variables
#    name     name of variable/function to find
# output : ndb  is the number of the package in which
#                                    variable is found.
#      jvar nonzero if name is a variable
#      jvar 0 (and function returns ERR) if not found
#      integer type code tc indicates variable type
#      parfind = OK if found
 integer ndb,jvar,tc,npack,parfind
 character*(*) name
```

The calling sequence for `rtfinder` is:

```
function rtfinder(name,jvar,ndb,tc,caller)
# input:
#   name    name of variable/function to find
# output :
#   ndb the number of the package in which
#       variable is found.
#   jvar    nonzero if name is a variable
#       0 (and function returns ERR) if not found
#   tc  integer indicating variable type
#   caller  a string used in the error message if
#                   variable not found.  suggested use
#                   is the name of the routine calling
#                   rtfinder.
#   rtfinder = OK if found
 integer ndb,jvar,tc,rtfinder
 character*(*) name, caller
```

73.2.2 Extracting Properties

Once you have found a variable, use `rtxdb` to get the address and size of the variable.

```
subroutine rtxdb(jvar, ndb, fwa, ndim, ilow, ihi, icol, access)
# get out facts about variable number jvar
# input:
#  jvar and ndb returned by parfind o:202:positive parenthesis level at end
        Unclosed open parenthesis at line 198
:202:positive parenthesis level at start of sectional division: reset to zer
        Unclosed open parenthesis at line 198
r rtfinder
#  access: access desired
#   (0=INFO only, 1= LIMITED, 2=FULL,-1=INFO_LIMITED)
# output:
#   fwa      address of variable
#   ilow, ihi    low, high subscripts
#   icol         column lengths in memory
#   ndim         number of dimensions
 integer jvar, ndb, fwa, ndim
 integer ilow(7),ihi(7),icol(7), access
```

Values for access `LIMITED` and `INFO_LIMITED` return the dimension information using a limiting string if present. `INFO` and `FULL` return the non-limited dimensioning information. The dimension information returned is the size the array currently occupies. If it is currently unallocated, `rtxdb` returns the size `allot` would allocate for it if it were called now.

If `rtxdb` is called with `access=FULL` or `LIMITED`, and if variable `autodyn` is `YES`, then `rtxdb` will first allocate storage for any unallocated array and then return the information as requested.

73.2.3 Changing a package name to a number

```
function glbpknum(pn)
#find number of package whose name is pn
integer glbpknum
character*(*) pn
```

73.3 Writing Attribute Services

Names known to the Basis Language, such as variable, function, or macro names, may have one or more attributes assigned to them. This can be done by an author using the variable description file, or done at runtime using the routine `rtcattr`. Routines are supplied for listing or editing every variable having a given set of attributes. This section describes how to write such routines.

The key element is the routine `rtserv`, which will evaluate an attribute expression and will call a user supplied subroutine for each macro, function, and/or variable name for which the given attribute expression is true. The determination of which type or types of names (macro, function,

variable) are evaluated, is under user control. NOTE: function and variable names are evaluated only if they exist in an initialized package.

The calling sequence is:

```
call rtserv(attr,actor,param,servestr,actstr)
```

where `attr` is a string containing the attribute expression to evaluate. If `attr` is " ", then the expression is always TRUE. (described more fully below).

`actor` is the name of a compiled subroutine (DO NOT put quotes around the name). The name `actor` must also be declared `external` in the routine that calls `rtserv`.

`param` is an integer scalar or array which will be passed to subroutine `actor`.

`servestr` is a string governing how and with what type of input the user-supplied server `actor` is called. (Described more fully below in section [Ref: `servestr`]).

`actstr` is a string determining which actions to perform on a name or temporary variable after it has been serviced by subroutine `actor`. (Temporary variables and string `actstr` are described more fully below (sections [Ref: temp-vars] and [Ref: `actstr`]).

73.3.1 Attribute Expressions

An attribute expression is a simple logical expression. In addition to attribute names, it can contain parentheses `( )`'s, and the operators & (and), `|` (or), and ~ (not). (An operator must always appear between attribute names). For example: if you wanted a server to be called with those names that have both attributes `a` and `b`, then use attribute expression `"a & b"`.

73.3.2 Servestr

SERVESTR is a string governing how and with what type of input the user supplied server `actor` is called.

SERVESTR consists of a `type_designator` followed by 0 or more blank delimited `keyword:value` combinations.

The `type_designator` is a string of 1 to 4 characters. This string can contain 1 or 0 instances of the letters "`m`", "`f`", "`v`", and "`p`", which stand for macro, function, variable, and package respectively. If its letter does not appear in the `type_designator`, then the server will be not called with any names of that type. If the letter does appear then the names of that type which satisfy the given attribute expression will be passed to the server routine `actor`.

The allowable keywords for SERVESTR and their default values are as follows. You do not have to specify a keyword if it is not applicable to your server, or if you wish to use the default value.

Keyword	Definition	Option values	Default value
serve	Whether the server is to be called with data, database index only, information only, or not called at all	`data`, `info`, `no`, `index`	`data`
skip	Whether the server is to not service any particular type of quantity	len0	none
pkg	Name of package where temporaries are to be created.	any package name	none
dims	Whether the dimension information returned reflects any SETLIMIT limitations set upon the variable.	`limited`, `unlimited`	`unlimited`
db	Whether one or all databases are to be available for servicing.	package name of the databases to service	all databases to be serviced.
lang	Language of the callback.	`fortran`, `c`	`fortran`

Examples follow at the end of this section.

Keyword `serve` determines what information about the names is passed to the server (or even if the server is to be called). If its value is `data`, then all information including the address to the data is passed. If (and only if) any macros or functions are to be serviced, then a temporary variable is created to hold the data, and the information passed refers to this temporary variable (including database indices, address, type, dimensionality information). Thus all information passed refers to the data. It should be noted that any function or macros served in this way will be invoked without arguments.

If the value of keyword `serve` is `info`, then the address to the data is not passed, no temporary variables are created, and the information passed (database indices, type, dimensions, etc.) refers to the name (not data). Thus if a name is a function, the information describes the function, not the data produced by the function.

If the keyword `serve` is set to `index`, then no information, other than database indices, is passed to the server. No temporary variables are created, since no data address is passed to the server. As in the `info` case, the indices passed reference the name (not necessarily data).

If the keyword `serve` is set to `no`, then the server is not called. This option is useful if you only want to perform an `ACTSTR` action on the names. In this case argument `actor` can be 0.

Keyword `skip` determines what quantities are not to be serviced. If this keyword is set to `len0` then no 0-length variables will be serviced, even if they satisfy the given attribute expression.

Keyword `pkg` MUST be specified if temporary variables might be created. This occurs only if keyword `serve` is set to `data` (the default value) and macros and/or functions are to be serviced, i.e. the `type_designator` contains an "`m`" or "`f`". If this is the case, set `pkg` to the name of the package where all temporary variable are to be created. Note: you can use package name `global`.

Keyword `dims` determines whether the dimensionality information passed to the server refers to

a variable as originally dimensioned, or if it reflects any limitations created by a limiting string. If `dims` is set to `unlimited` then the former is given, else if `limited` the latter is given. The default is `unlimited`.

Keyword `db` determines if one or all databases are to be serviced by subroutine `actor`. If this keyword is not present in SERVESTR, then all databases (and macros if requested) are serviced. Otherwise, you can set keyword `db` to the name of a package, in which case, the named package is the only package serviced. It should be noted that if `db` is set, then macros can not be serviced. If you are interested in the global database, then set `db` to `global`.

```
EXAMPLES:
      "mfv pkg:tmp dims:limited"
      "mfv serve:info"
      "v"
      "v skip:len0 db:global"
```

The first SERVESTR will service macros, functions, and variables. The data address is passed and temporary variables will be placed in package `TMP`. The dimensionality information will refer to the limited portion of the data.

The second SERVESTR will service macros, functions, and variables, but the data address is not passed and no temporary variable are created. The information returned refers to the name and the dimensions returned describe a variable as originally dimensioned. Functions and macros have no dimensions.

The third SERVESTR services only variables. The data address is passed along the with all other information, including the dimensions of the variable as originally declared.

The fourth SERVESTR services only global variables which are not of 0-length. The data address is passed along with all other information, including the dimensions of the variable as originally declared.

73.3.3 Actstr

ACTSTR is a string determining which actions to perform on a name or temporary after it has been serviced by subroutine `actor`.

If ACTSTR is `" "` then no additional actions are performed. Otherwise ACTSTR is a series of 0 or more blank delimited keyword:value combinations.

The allowable keywords for ACTSTR and their default values are as follows. If a keyword is not specified then, the action corresponding to that keyword is not performed.

Keywords	Definition	Option values
forget	forget any temporaries created and/or the original name	`name`, `temp`, `all`
tag	tag any temporaries and/or the original name with the given attribute list `ATTLIST`	`name: ATTLIST`, `temp: ATTLIST`, `all: ATTLIST`

Action keywords `forget` and `tag` have three possible values: `name`, `temp`, and `all` which causes the action to be performed on the the original name, on any temporary variable which may have been created, or on both the original and temporary variable, respectively.

The action `forget` will cause the names and/or temporaries to be forgotten. The action `tag` will cause names and/or temporaries to be tagged with a given set of attributes (attributes may also be forgotten). Thus a third component of `tag` action is an attribute list, which is a list of attributes names separated by blanks, `+`, or `-`. A blank or `+` preceding an attribute means to add this attribute; if prefixed by a `-`, then the attribute is removed. Note: the attribute list must be written without blanks.

All actions are performed after the name has been serviced.

EXAMPLES:

```
"forget:name tag:temp:myatt"
            ## pkg: option was given in SERVESTR
"tag:name:myatt-oldatt"
"forget:all"
" "
```

The first `ACTSTR` will cause all the original names serviced to be forgotten, and all the temporary variables (in package `TMP`) to be tagged with the attribute `MYATT`.

The second `ACTSTR` will cause all the original names serviced to be tagged with the attribute `MYATT` and to remove the attribute `OLDATT`.

The third argument will cause both the original names and the temporary variables to be forgotten. It is assumed that keyword `pkg` was set in `SETSTR`. If not, then an error occurs.

The fourth string will not perform any actions.

73.3.4 RTSERV and Temporary Variables

You will most likely want to tag or forget any temporary variables which were created and also add a new group to the end of the package vdf file in which the temporaries are to be stored. The reasons for this are described below.

It is safest if a special group exists which is dedicated to holding the generated temporary variables. This group MUST be the last group in your package vdf file.

Efficiency comes into play when you are servicing data created by functions and macros. You will eventually want to forget (i.e. destroy) all the temporary variables which were created (in order to reclaim the space). However, if you need to reference this data over multiple calls to RTSERV you may not want to create and destroy the temporary variables for each RTSERV call. You can avoid this by tagging these temporary variables with two or more attributes: one attribute to mark them for future deletion and the other attribute(s) to allow future servicing.

The following example demonstrates this method.

NOTE: the order of the variables might change between the server call which creates the temporaries and the next server call which uses those temporaries.

EXAMPLES:

```
call rtserv("myatr", myserv1, param,
                          "mfv pkg:tmp",
                          "tag:temp:myatr+tempv")
call rtserv("myatr", myserv2,
                          param, "v", " ")
call rtserv("tempv", 0, param, "v serve:no",
                          "forget:name")
```

In the above example, it is assumed that attribute TEMPV is used only to tag variables for deletion.

The first call to RTSERV services all macro, functions, and variables with attribute MYATR. The temporary variables are then tagged for later servicing and deletion. The second call to RTSERV services only variables with attribute MYATR. It will find the variable data generated by the macros and functions of the first call to RTSERV, since it was not destroyed and was marked with attribute MYATR. The third call to RTSERV will destroy the macro and function data generated by the first call, since this data was tagged with attribute TEMPV. Notice that keyword `serve` is set to `no` in order to improve efficiency.

As noted above, the order in which the variables are served may change in the above example between the server calls `myserv1` and `myserv2`, due to the creation of temporary variables between these two calls. If it is important that the ordering does not change, then you can do an extra `rtserv` call that does nothing except create the temporaries and change the ordering so that the ordering would remain constant for all subsequent calls. The previous example would be modified as follows:

EXAMPLES:

```
call rtserv("myatr", rtcount, param,
                              "mf pkg:tmp",
                              "tag:temp:myatr+tempv")
call rtserv("myatr", myserv1, param, "v", " ")
call rtserv("myatr", myserv2, param, "v", " ")
call rtserv("tempv", 0, param, "v serve:no",
                               "forget:name")
```

Note: rtcount is a Basis supplied server which returns the number of entities (variables, macros, functions) serviced.

The user-supplied attribute server will be called in four stages. First, rtserv calls actor with argument stage set to 0. Next, if packages are selected actor is called with stage set to 3. Then, for each name satisfying the attribute expression attr, rtserv calls actor with argument stage set to 1. [NOTE: your user server will not be called with any name that resides in an uninitialized package.]. Finally, when all processing is complete, rtserv calls actor with argument stage set to 2.

The fortran interface of subroutine actor should be of the form:

```
call  actor(npack,jvar,name,typecode,fwa,ndim,ilow,
                  ihi,icol,attr,param,moreargs,istage)
```

where npack is the number of the database package, jvar is the index into database npack, name is the name of the variable, typecode is an integer giving the type of the variable, fwa is the first-word-address of the variable, ndim is the number of dimensions, ilow is an array of up to 7 integers containing the origin subscript in each dimension, ihi an array of up to 7 integers containing the highest subscript in each dimension, and icol an array of up to 7 integers containing the column length in each dimension. The value of param is passed through from rtserv and istage is set by rtserv, above.

The value of argument moreargs is integer data passed down from subroutine rtserv. It is available to supply the user with addition information about the name or about the server options (i.e. servestr) selected in rtserv. Currently only 1 value of moreargs is defined. moreargs (1) is set to 0 if name is a variable, 1 if name is a function, or 2 if name is a macro.

It should be noted that if the rtserv call has SERVESTR keyword serve set to info then actor argument fwa is not set. If keyword serve is set to index, then actor arguments fwa, typecode, ndim, ilow, ihi, and icol are not set.

More esoteric note: If keyword serve is set to either info or index, and name is a macro, then npack is set to 0 and jvar is set to the macro number. If you wish to use these numbers, you must call special macro routines, and NOT the standard Basis database routines.

The C interface of subroutine actor should be of the form:

```
void actor(BA_dbnode *node, void *param, int istage)
```

where node is a pointer a database node, either macro, variable or function. param is a pointer to user data and istage is the stage.

73.4 Basis Supplied Servers

In addition to writing your own servers, there are currently two servers available in Basis which you can supply to rtserv. They are rtcount and rtcntsiz. They both have the standard server interface which is

```
call actor(npack,jvar,name,typecode,fwa,ndim,ilow,
                 ihi,icol,attr,param,moreargs,istage)
```

Both servers return output in argument `param`. In server `rtcount`, `param(1)` is set to the number of entities (variables, macros, functions) which the server was called with. Server `rtcntsiz` is an extension of server `rtcount`. In addition to the number of entities, it also returns the total number of words of data for those entities, and an error flag. `param(1)` is set to the number of entities, `param(2)` is set to the total data length, and `param(3)` is the error flag which is set to 1 if an error occurred, otherwise it is set to 0. An error occurs if the data dimension information is not available, such as a macro or function for which a temporary variable has not been made or if `servestr` option `serve` have been set to `index`.

73.5 Writing Built-in Functions

It is possible to write built-in functions which the Basis parser knows about. To do this, you need to do two things. You need to write a subroutine `pkgbfcn`, where `pkg` is the name of the package containing the built-in functions, and you need to add a declaration of these functions to the variable descriptor file.

The way to declare a built-in function into a variable descriptor file is described in detail in section [Ref: wrpkgs-fxns] “Functions” in chapter [Ref: wrpkgs] “Writing Basis Packages”. Package `bes`, described in the chapter “Basis Package Library”, is a very simple example of writing a built-in function. The following is a descriptor file used to declare in package `tst` the built-in function variable `mydummy`, a function which takes 1–3 arguments.

```
tst
### variable descriptor file for package tst
### this package illustrates how to add built-in function
### mydummy.
***** Dummy_1:
mydummy(array [,ilen [,idim]]) builtin [1-3]
# Reduce the length of dimension idim in array by ilen
# default value for ilen = 0.
# default value for idim = last dimension.
# This function has no purpose other than
# illustrating how to install built-in functions.
```

Additional parameters used by the subroutine are

`ERR` Value returned if an error occurred.

`OK` Value returned if no error occurred.

ERR and OK are defined automatically by MAC.

Note that all these parameters must be in CAPITAL letters.

The subroutine to execute the built-in functions needs to call a number of Basis functions and subroutines. These functions will be described and then a sample subroutine will be provided which will execute built-in function mydummy in package tst. When using these routines, you must spell their names exactly as seen below. There is a difference between spelling a name in UPPER, lower, or Mixed case.

Dynamic, Point, remark

```
Dynamic(name, type, ndim)
Point(arraynam)
call remark(string)
```

These routines have been previously described—Dynamic and Point in section [Ref: dynamic-dimensioning] “Dynamic Dimensioning” and remark in section [Ref: output-routines] “Output Routines”. In brief, Dynamic is used to declare dynamic arrays. The macro Point is used in conjunction with paraddr (described later) to equivalence array arraynam to the data of an argument. Subroutine remark is used to print messages to the terminal.

When a built-in function is called, the Basis parser creates data descriptors of all of its arguments on the parser stack. In order to access these arguments, you will need to use the following two functions:

arg_fetch_init `call arg_fetch_init(nargs, sx)` Initialization function to allow fortran to access stack variables. nargs is the number of arguments and sx is the return value. They will be passed to you through the pkgbfcn function call.

arg_fetch_fin `call arg_fetch_fin()` Call after all processing is finished.

Once the arguments are initialized, they can be fetched with these routines.

arg_fetch_actual `arg_fetch_actual(iarg)`

arg_fetch_copy `arg_fetch_copy(iarg)`

arg_fetch_default `arg_fetch_default(n, iarg, name)`

Once the arguments have been fetched we can use the following routines to get more information about the data.

arg_get_address `pointer = arg_get_address(iarg)` returns the pointer to the data of argument iarg. It is used in conjunction with macro Point to equivalence an array to this data. Remember to declare paraddr of type Address. This was paraddr(dd).

arg_get_name `len = arg_get_name(iarg, name)` set `name` to the name of argument `iarg`.

arg_get_type `tc = arg_get_type(iarg)`

arg_get_shape `arg_get_shape(iarg, extent, lower, stride)`

arg_fix_dim `arg_fix_dim(iarg)`

arg_get_length `arg_get_length(iarg)` returns the length of the data in argument `iarg`. Remember to declare `arg_get_length` of type integer. This was `parlen(dd)`.

arg_get_integer `if (arg_get_integer(iarg, mying) .eq. ERR) return (ERR)` if you expect some data to be a scalar integer value, then you may like to call function `arg_get_integer`. This function will check if the data of argument `iarg` is a scalar integer. If it is, then this routine will set `myint` to this integer value and the function will return the parametrized value `OK`. Otherwise the routine will print an error message and return the parametrized value `ERR`. Remember to declare `parint` of type integer. This was `parint(dd, myint)`.

arg_get_coerce `if (arg_coerce(iarg, tc) .eq. ERR) return (ERR)` will coerce the data of argument `iarg` to the type `tc` and modify the data descriptor to reflect the change. If `iarg` cannot be coerced to type `tc`, then this function will print an error message and return the value `ERR`. Otherwise the routine will return the value `OK`. Remember to declare `parcoerc` of type integer. This was `parcoerc(dd, tc)`.

arg_kill `call arg_kill(iarg)` releases the memory of the data of argument `iarg` Used to release the memory of all the input arguments of a built-in function. This must be done in order to avoid memory leaks. This was `parrel(dd)`.

utstrcod typecode = utstrcod(typestr)returns the type code associated to `typestr`. Valid values for `typestr` are `"integer"`, `"real"`, `"external"`, `"name"`, `"complex"`, `"logical"`, `"chameleon"`, `"indirect"`, `"group"`, `"double"`, `"structure"`, `"range"`, `"function"`, `"address"`, `"string"`, and `"null"`. Remember to declare `utstrcod` to be type integer.

Finally, to create the return value we use these routines. `sx` is the value passed into `pkgbfcn`.

sx_set_ndim `sx_set_ndim(sx, ndim)` sets the number of dimensions of `sx` to `ndim`.

sx_set_type `sx_set_type(sx, tc)` sets the type `sx` to `tc`.

sx_set_shape `sx_set_shape(sx, extent, lower, stride)` Sets the shape of `sx` to `extent`, `lower`, and `stride`. Each must be an array at least `sx_get_ndim` long.

parget `call parget(sx)` gets enough space to hold all of the data described by the data descriptor `sx`. Note: this routine does not store or retrieve any data. It just gets the required space. `arg_get_address(0)` can be used to find the address.

73.5.1 Sample Subroutine PKGBFCN

To install a built-in function you must write a function called `pkgbfcn` where `pkg` is the name of your package. This subroutine is described as follows:

```
function pkgbfcn(nargs, f, sx)
```

`nargs` number of arguments built-in function was called with

`f` name of built-in function

`sx` (output) data descriptor of built-in function's output

Argument `sx` is the only output argument. It is the data descriptor which describes the output returned by the built-in function. Your job is to determine the type and size of `sx`, set corresponding entries of `sx`, call `parget(sx)` to get storage, then fill the storage with the result.

A sample `pkgbfcn` subroutine follows. The subroutine's name is `tstbfcn` since the built-in function `mydummy` is declared in package `tst`.

```
function tstbfcn(nargs,f,sx)
implicit none
integer nargs, tstbfcn
character*(*) f    #name of function
integer sx

integer inttyp   ! type code for integer
integer realtyp  ! type code for real
integer cmplxtyp ! type code for complex
integer tc       ! type code
integer i, idim, ndim, ilen, nskip, nstore, npoints, indxx, indxy
integer extent(7), lower(7), stride(7)
integer, external :: utstrcod  ! converts type string to a type code
external parget  ! gets space for an element
integer, external :: arg_get_type
integer, external :: arg_get_integer ! gets integer
external :: arg_get_shape
integer, external :: arg_get_ndim
integer, external :: arg_get_length  ! gets the length of a stack ele
Address, external :: arg_get_address ! gets pointer to data
Dynamic(ix,integer,1)    ! integer output argument
Dynamic(rx, real, 1)     ! real output argument
Dynamic(cx,complex,1)    ! complex output argument
Dynamic(iy, integer, 1)  ! integer input argument1
Dynamic(ry, real, 1)     ! real input argument1
```

```
      Dynamic(cy, complex, 1)  ! complex input argument1

!   mydummy (arrayname [, ilen [, idim]])
!   arrayname type can be integer, real, or complex
!   default value for ilen is 0
!   default value for idim is sy(SS_N) i.e. last dimension

      tstbfcn = ERR
      call arg_fetch_init(nargs, sx)

      if (f .eq. "mydummy") then
         ! get the type codes for the types integer, real and complex.
         inttyp = utstrcod ("integer")
         realtyp = utstrcod ("real")
         cmplxtyp = utstrcod ("complex")

         ! get the first argument
         call arg_fetch_copy(1)
         ndim = arg_get_ndim(1);

         ! the second argument (if present) must be an integer scalar.
         ! store its value into ilen
         if (nargs >= 2) then
            ! call fcnargb(2, sz)   ! get second argument
            call arg_fetch_copy(2)
            if (arg_get_integer(2, ilen) .eq. ERR) return
            if (ilen < 0) then
               call remark("mydummy:  arg2 is negative")
               return
            endif
         else
            ilen = 0
         endif

         ! the third argument (if present) must be an integer scalar.
         ! store its value into idim
         if (nargs = 3) then
            call arg_fetch_copy(3)   ! get third argument
            if (arg_get_integer(3, idim) .eq. ERR) return
            if (idim < 0 | idim > ndim) then
               call remark("mydummy:  arg3 is out of range")
               return
            endif
         else
            idim = ndim
```

```
endif

! shape, size, and type of the output is almost the same as
!   the input's.
! reset shape and size of output as follows:
! the length of dimension idim in the output array is ilen
! shorter than that dimension in the input array
! Make sure the new length of that dimension is still positive
tc = arg_get_type(1)
call sx_set_type(sx, tc)

call arg_get_shape(1, extent, lower, stride)
extent(idim) = extent(idim) - ilen
! reset one entry of sx
if (extent(idim) .le. 0) then
   call remark("mydummy:  arg2 is too large")
   return
endif
call sx_set_ndim(sx, ndim)
call sx_set_shape(sx, extent, lower, stride)

! calculate the number of consecutive elements in the input to
! be stored into the output --- nstore
! calculate the number of consecutive elements in the input
! which are not stored into the output --- nskip
nskip = 1
do i = 1, idim-1
   nskip = nskip*extent(i)
enddo
nstore = lower(idim)*nskip
nskip = ilen*nskip

! get space for answer
call parget(sx)
npoints = arg_get_length(1)         ! size of input array

if (tc .eq. inttyp) then
   ! store integer output
   Point(ix) = arg_get_address(0)   ! ix is integer output array
   Point(iy) = arg_get_address(1)   ! iy is integer input array
   indxx = 1
   indxy = 1
   do
      do i = 1, nstore
         ix(indxx) = iy(indxy)
```

```
               indxx = indxx+1
               indxy = indxy+1
            enddo
            indxy = indxy + nskip
            if (indxy .gt. npoints) exit
         enddo

       elseif (tc == realtyp) then
          ! store real output
          Point(rx) = arg_get_address(0)   ! ix is real output array
          Point(ry) = arg_get_address(1)   ! iy is real input array
          indxx = 1
          indxy = 1
          do
             do i = 1, nstore
                rx(indxx) = ry(indxy)
                indxx = indxx+1
                indxy = indxy+1
             enddo
             indxy = indxy + nskip
             if (indxy .gt. npoints) exit
          enddo

      elseif (tc == cmplxtyp) then
         ! store complex output
         Point(cx) = arg_get_address(0)   ! cx is complex output array
         Point(cy) = arg_get_address(1)   ! iy is complex input array
         indxx = 1
         indxy = 1
         do
            do i = 1, nstore
               cx(indxx) = cy(indxy)
               indxx = indxx+1
               indxy = indxy+1
            enddo
            indxy = indxy + nskip
            if (indxy .gt. npoints) exit
         enddo

      else
         call remark("mydummy: arg1 is wrong type")
         return
      endif
   else
      call remark("unknown type for built-in function mydummy")
```

```
   return
endif

! release storage occupied by the input
call arg_kill(1)
if (nargs >= 2) call arg_kill(2)
if (nargs >= 3) call arg_kill(3)
call arg_fetch_fin
tstbfcn = OK
return
end
```

73.6 Foreign Packages

73.6.1 Cooperating with Other Systems

A foreign package is created by using the keyword "`foreign`" instead of the word "`package`" in the `GLUEPACK` input file. The effect of this declaration is to require additional routines to be written by the author. These routines are to communicate with Basis about the attributes of variables which are NOT listed in the variable description file.

The foreign-package facility allows you to write a package which creates variables that did not exist at compile time and wishes to make these variables known to Basis. Or, you may write a package that uses some symbolic memory manager to manage some variables and Basis needs to access them too.

Only some services are available for foreign variables. Basis can use them or assign to them. Basis cannot change their size, `FORGET` them, and in `LIST`ing them, Basis only knows their basic properties, not things like their original dimensioning string and units.

A foreign package is otherwise identical to a regular package. In particular it has a variable description file (which is perhaps nearly empty) and must be connected using `GLUEPACK`.

73.6.2 The Foreign Connection

A foreign package must supply three extra routines. These have names `pkgfind`, `pkgxdb`, and `pkgxcom`, where `pkg` is the name of the package.

```
function pkgfind( name, typecode)
character*(*) name
integer key, typecode, pkgfind
```

is a function which takes input `name` and returns a positive integer function value and an integer type code `typecode`. The function value should be 0 if `name` is unknown. Otherwise, it should

be set to a positive integer (whose meaning is up to you) and `typecode` should be set to a code which gives the type of the variable `name`. This type code can be obtained with the utility function `utstrcod` which is an integer function taking a string as an argument and returning the corresponding code, such as

```
typecode = utstrcod("integer")
typecode = utstrcod("real")
typecode = utstrcod("character*(12)")
```

It will be faster, of course, to get the typecodes you will need once during the package initialization routine `pkginit`. After `pkgfind` returns successfully, the other two routines may be called by Basis. Basis will pass the function value you return from `pkgfind` back to you as the argument named `key`.

```
subroutine pkgxdb(key, fwa, ndim, ilow, ihi, icol)
integer key, fwa, ndim, ilow(7), ihi(7), icol(7)
```

This function must return, for the variable last found with `pkgfind`, the address (`fwa`), the number of dimensions (`ndim`), and the first `ndim` entries of `ilow`, `ihi`, and `icol`, giving respectively the lowest subscript for the variable (usually 1), the highest subscript (usually the length in that dimension), and the dimension length in memory (usually the length in that dimension, but possibly not, such as a matrix which is only partially full). For example, if the variable is dimensioned (0:10,12) but currently contains meaningful elements in the first 8 rows and the first 5 columns, `pkgxdb` would return:

```
fwa = loc of first element
ndim = 2
ilow(1) = 0
ihi(1) = 7   #highest meaningful subscript
icol(1) = 11
ilow(2) = 1
ihi(2) = 5   #highest meaningful subscript
icol(2) = 12
```

The third routine allows Basis to process any comments available for the variables.

```
subroutine pkgxcom(key, icom, comment)
integer key, icom
character*(*) comment
```

This routine has `key` and `icom` as input. The value of `icom` will be 1 the first time, and then increase by 1 with each subsequent call. The output `comment` should be set to the `icom`'th comment line available for the variable. If no such comment is available, comment should be set to all blanks with

```
comment = " "
```

Fortran blank-fills in character assignment statements so the statement above sets all of `comment` to blanks.

Sometimes Basis will ask only for the first comment. Other times Basis will ask for successive comments until it gets a blank comment back. If you don't wish to supply online documentation for the variables, it suffices to have `pkgxcom` simply set `comment` to blank and return.

The routine `pkginit` is an appropriate place to initialize your memory manager or other tables.

73.6.3 Sample Foreign Package

Here are some pieces of a sample foreign package named `lsp`. (This is a modification of the full sample package presented later.) The package contains five variables named `x`, `y`, `z`, `w`, and `wc`, which are made visible to Basis. A table of their properties is initialized in lspinit.

Variable Description File

The variable description file is the same as usual; it contains definitions for some variables and one function. We have chosen to use it to declare the variables needed for the symbol tables for the foreign variables.

```
lsp
#This package illustrates how to write a Fortran driver for lsode
#lsode calls a user function in Basis for its values.
{
NFRGN = 5  # size of foreign variable tables
NFRGN1 = NFRGN + 1
MAXCOMMENTS = 50
}
****** Lsodet:
userfn        Varname
   #name of basis function to be called by lsode
lrw    integer /0/
   #length of real work area
rwork(lrw)    _real
   #dynamic storage for real work area
liw    integer /0/
   #length of integer work area
iwork(liw)    _integer
   #dynamic storage for integer work area
neqc        integer /0/
   #number of equations to be solved
```

```
yc(neqc)    _real
   #current values of solution
tc    real
   #current value of time
ydotc(neqc)    _real
   #place for function to put the values it computes
**** Functions:
lsdriver(f:string,neq:integer,y,t,tout,rtol,atol:real) subroutine
   #makes this compiled routine visible to Basis

***** Tables hidden:
# Tables to hold attributes of Foreign variables
$ These tables don't need to be in the variable description file
$ but they are in this example.
$ This group can be declared hidden so the user won't see it.
names(NFRGN)   Varname  #names of variables
tcs(NFRGN)     integer  #types
ndims(NFRGN)   integer  #dimensions
ilows(7,NFRGN) integer  #low subscripts
ihis(7,NFRGN)  integer  #high subscripts
fwas(NFRGN)    integer  #addresses
comments(MAXCOMMENTS) character*72 #comments about variables
cfirst(NFRGN1) integer  # points into comments
```

Configure File

It is the word "`foreign`" on line 1 of this file that makes the package foreign. Having done that, we need to supply the routines `lspfind`, `lspxdb`, and `lspxcom`. As line 1 indicates, we shall also need to supply `lspinit`.

```
     foreign  , lsp = "Interactive Lsode" , limit = 10, init
     codename = Lsode , firstpkg = lsp , cprompt = 'Lsode> '
     macfile = lspbas , probname = lout , codefile = lsode
```

Foreign Connections in the Source File

Here are the four routines used to implement the foreign variables. We have chosen to initialize the tables in `lspinit`.

```
      subroutine lspinit
# initialize tables for foreign package calls
Use(Tables)
### these are the variables which are "foreign" ##########
```

```
      real x
      integer y
      real z(10)
      real w(5,-3:3)
      character*12 wc
      common /lspa/ x,y,z,w
      common /lspc/ wc
#######################################################
      integer icom
      integer utstrcod
      external utstrcod
# size information
      data ndims/0,0,1,2,0/
      data ilows(1,3) / 1 /
      data ihis(1,3) /10/
      data ilows(1,4) /1/
      data ihis(1,4)/5/
      data ilows(2,4) /-3/
      data ihis(2,4) /3/
# names
      names(1) = "x"
      names(2) = "y"
      names(3) = "z"
      names(4) = "w"
      names(5) = "wc"

# types
      tcs(1) = utstrcod("real")
      tcs(2) = utstrcod("integer")
      tcs(3) = utstrcod("real")
      tcs(4) = utstrcod("real")
      tcs(5) = utstrcod("character*(12)")
# addresses
      fwas(1) = loc(x)
      fwas(2) = loc(y)
      fwas(3) = loc(z)
      fwas(4) = loc(w)
      fwas(5) = loc(wc)
# comments
# cfirst(i) to cfirst(i+1) -1 are the comments for i'th entry.
# use icom to make it easy to add new comments.
# caution, no checking done on overflowing comments array.
      icom = 1
      cfirst(1) = icom
      comments(icom) = "Facts about x" ; icom = icom + 1
```

```
      comments(icom) = "Comments about x are hard to come by."
      icom = icom + 1
      comments(icom) = "Perhaps we should investigate."
      icom = icom + 1
      cfirst(2) = icom
      comments(icom) =
      "Little is known about y except that she likes flowers."
      icom = icom + 1
      cfirst(3) = icom

      comments(icom) = "Little known about z"
      icom = icom + 1
      comments(icom) = "except he once lived in Indiana"
      icom = icom + 1
      cfirst(4) = icom
      comments(icom) = "w is two-d as you can see"
      icom = icom + 1
      cfirst(5) = icom
      comments(icom) = "Say the secret word and win 50 dollars"
      icom = icom + 1
      cfirst(6) = icom
      return
      end
      function lspfind(name,tc)
# given name, return key as function value
# return 0 if not found
# if found, return type code tc
      integer tc, lspfind
      character*(*) name
Use(Tables)
      do i=1, NFRGN
 if(name = names(i)) then
     tc = tcs(i)
     return(i)
  endif
      enddo
      return(0)
      end

      subroutine lspxdb(jvar,fwa,ndim,ilow,ihi,icol)
# given key jvar returned by lspfind, return address (fwa),
# dimension (ndim), low subscripts (ilow), high subscripts (ihi),
# and dimensions in memory (icol)
      integer jvar,fwa,ndim,ilow(*), ihi(*),icol(*)
      character*(*) name
```

```
Use(Tables)
      if( jvar < 1 | jvar > NFRGN) call baderr("lspxdb error")
      fwa = fwas(jvar)
      ndim = ndims(jvar)
      do j=1, ndim
 ilow(j) = ilows(j,jvar)
 ihi(j)  = ihis(j,jvar)
 icol(j) = ihis(j,jvar) - ilows(j,jvar) + 1
      enddo
      return
      end

      subroutine lspxcom(jvar,icom,comment)
      integer jvar, icom, jcom
      character*(*) comment
# returns the icom'th comment about the variable
#                                  whose key is jvar
Use(Tables)
      if( jvar < 1 | jvar > NFRGN)
         call baderr("lspxcom error")
      jcom = cfirst(jvar) + (icom - 1)
      if( jcom < cfirst(jvar+1)) then
 comment = comments(jcom)
      else
 comment = " "
      endif
      return
      end
```

Part VI

The Basis Package Library

CHAPTER

SEVENTYFOUR

Basis Package Library

This manual contains short descriptions of packages available for inclusion in your program. To include one of these packages in your program, you simply include its name in your directory list to mmm, and mmm takes care of the rest.

The source for these packages is available in the Basis source distribution as subdirectories of the library directory. The naming conventions followed in most of them are:

- pkg.m is the MPPL sources.
- pkg.v is the variable description file.
- pkg.pack is a CONFIG input file describing the package.
- pkg.doc is a text file telling how to use the package.
- pkg.in is a Basis Language input file that the package reads when it is initialized. This file often does not exist.
- mmm control files are provided so that the package can be compiled with the `mmm` utility.

The binaries for the packages are installed in $BASIS_ROOT/lib, and their pack file is in $BASIS_ROOT/include.

CHAPTER

SEVENTYFIVE

BES: Bessel Functions

`bes` is a package providing a few Bessel functions as built-ins. This package is also a very simple example of writing built-in functions.

```
Author: Bruce Langdon, Version 0, 5/89
        Kimberly  Anderson, Version 1, 6/90

Usage:

i0(x), i1(x), k0(x), k1(x)
        with x a real scalar or vector,
        return the values of the modified Bessel
        functions of order zero and one.

j0(x), y0(x), j1(x), y1(x)
with x a real scalar of vector, return the values of the Bessel
functions of order zero and one.
```

The error tolerance on all these functions (as found by comparison to Abramowitz and Stegun tables) is about 1E-7. -

CHAPTER

SEVENTYSIX

CTL: Package Control

76.1 The History of The CTL Package

When Basis was first written, it did not yet include the ability to call compiled functions from the Basis Language. In order to be able to run programs while we figured out how to accomplish the goal of calling compiled functions, a simple model was devised and built into Basis so that a user could issue the commands run, generate, step, and finish to control the basic parts of the simulation. Later, this model was removed from Basis proper and made into this CTL package to provide the facility to older programs that still needed it. Obsolete though it is in some sense, people have continued to use this package because it fits many programs exactly, so we continue to support it despite the complication it adds to the config program.

76.2 The CTL Model

This package is meant to be used in conjunction with other packages. It supplies the command `run`, with subsidiary commands `generate`, `step`, and `finish` for more detailed control. The next section describes the `ctl` generate-step-finish model. Subsequent sections describe how to use the commands, and how to install `ctl` into a program.

76.3 The CTL Model

Using the `ctl` model, each package has six executable sections:

1. Generator.
2. Generator plots.
3. Execute a “step.”
4. Post-step plots.

```
(Insert Package Execution Model
graphic illustration here.)
```

Figure 76.1: Package Execution Model

5. Finish (final edits, etc.).
6. Finish plots.

Normally, a package would be run by executing steps 1 and 2, repeating steps 3 and 4 until the problem is completed, then finally executing steps 5 and 6. The `run` command does just this, with optional disabling of plots and an optional limit to the number of times the step is executed. Of course, not all packages have active modules in all of these places. For example, there may be no step part at all, or it may always complete in one step. By using the `generate`, `step`, and `finish` commands, the user can control the six parts in some detail. The generator must be executed before any of the others, however.

76.4 The User Interface

The user interface supplied with `ctl` consists of variables the user can set plus the commands `generate`, `step maxsteps`, `finish`, and `run maxsteps`. The command `run(maxsteps)` is equivalent to:

```
generate
step(maxsteps)
finish
```

Each of the other three commands drives the corresponding section of the model. The optional argument `maxsteps` to the `step` command can be used to set a maximum number of steps to be taken before returning. Each of the commands sets the variable `ctlstat` with the value of the status returned by each step: 0 = completed O.K., -1 = error. The `step` command may also return the value 1 = DONE, indicating the package has concluded its “step” phase.

The detailed behavior of the commands can be changed by setting certain variables in the `ctl` package. These are:

- ctlpkg – the name of the package to run. If blank, the default, the current package is used.
- ctlplot – if `no`, do not run the stages `pkggenp`, `pkgexep`, `pkgfinp`.
- ctlexe – if `no`, do not run the stages `pkggen`, `pkgexe`, and `pkgfin`.
- ctlopt, nctlopt – ctlopt is an array of 32 integers, which can be set by the user. The values ctlopt, nctlopt are used as arguments to each of the six stage routines. The default value of nctlopt is 0.

76.5 Adding CTL to Your Program

This section contains instructions for authors on how to add the `ctl` package to their program.

76.5.1 Using the Model

Each of the six stages is driven by a separate routine. You will write some or all of these six routines according to the specifications below. Then you will include file `ctl.pack` in your CONFIG input, and also inform CONFIG in your descriptions of other packages of which of the six routines you have written.

Deciding how to divide your calculation between the six functions `pkggen`, `pkggenp`, `pkgexe`, `pkgexep`, `pkgfin`, and `pkgfinp` is an important step. You can do plotting in any of the six steps. The user is then going to be able to run or not run the "p"-suffixed routines by setting control variables in `ctl`. For example, "ctlplot=no;run" skips all plotting routines and results in calls to pkggen, pkgexe (iteratively), and pkgfin only. It may be appropriate to do some plots no matter what the user enters; this is entirely up to you. Generally you will want to confine plotting to the "p" routines and to use the iteration loop if at all appropriate. Which plotting packages you use are up to you.

76.5.2 Connecting Everything Up

The routines shown in the model routines are called by `ctl`. The CONFIG program supplies "calls" to any of these routines that apply to your case. In subroutine and function names, replace the letters pkg with your package name (i.e., myinit, mygen, mygenp, ..., myvers). Or, you may supply your own names for these routines; see the section "Configuring the Packages" in manual V, "Writing Basis Programs" for how to do this. In what follows, we will refer to these routines in the form "`pkgrout`" (where "`rout`" is the root name of the routine, such as `gen`, etc.), but bear in mind that you may give them your own names.

For each of the six routines that you do supply, include the root name of the routine in the description of the corresponding package in the CONFIG input file. For example, if you have a package named `abc` and you choose to write `abcgen` and `abcexe`, then you would put the words `gen` and `exe` in the CONFIG input file, such as:

```
package abc="ABC algorithm" gen exe limit=100
package ctl="Control Package"
firstpkg=(abc,ctl)
```

76.5.3 Passing Options

The six routines each have the arguments optlist,nopt. These should be declared:

```
integer optlist(32), nopt
```

The user may set the variables `ctlopt` and `nctlopt` and the user commands pass these values to the routines. Authors may make whatever use they wish of these.

76.5.4 Functions PKGGEN and PKGGENP

The `generate` command calls the function `pkggen(optlist,nopt)` to "generate" a problem, typically after the user has set parameters using the parser. What "generate" means is up to you. Typically you will want to use `pkggen` to do problem-dependent initialization, and for packages which have no iteration loop, `pkggen` may be the only working module. Basis calls the function `pkggenp(optlist,nopt)` after `pkggen` and does any plots desired after `pkggen` has executed. Note that one cannot ensure that `pkggenp` will ever be executed since the user may turn plotting off. However, one can be sure that `pkggen` will be executed before either `pkgexe` or `pkgfin`, described below.

You must declare `pkggen` and `pkggenp` to be type integer and they must return the symbolic integers OK or ERR to indicate success and failure (CASE COUNTS).

76.5.5 Functions PKGEXE and PKGEXEP

The `step` command calls function `pkgexe(optlist,nopt)` repeatedly until it returns either DONE or ERR. It returns DONE when the problem has been completed, ERR if an error has occurred, and OK if it should be called again. The step command may be given an integer argument indicating the maximum number of steps to be taken. If the argument is not supplied, the value defaults to the one set by CONFIG.

After each completion of pkgexe that returns OK or DONE, Basis calls `pkgexep(optlist,nopt)` to do plots requested at that time. What `pkgexe` does on each call is entirely up to the package author: a step in time, a trace of one ray among several, etc. The functions `pkgexe` and `pkgexep` must be declared type integer.

76.5.6 Functions PKGFIN and PKGFINP

Finally, Basis calls the two functions to do final edits and plots at the completion of a run, `pkgfin(optlist,nopt)` and `pkgfinp(optlist,nopt)`. Any other action desired can be put in these routines. Basis allows the users to run these two routines together or separately at any stage of the calculation, so they should be designed accordingly. These functions must be type integer and return OK or ERR. -

CHAPTER

SEVENTYSEVEN

FFT: Fast Fourier Transforms

77.1 Routine Interfaces

The FFT package consists of two functions that implement Fast Fourier Transforms:

- fft(x;dim) returns the discrete Fourier transform of real or complex array x. If present, dim is the dimension over which the transform is taken for all values of the other subscripts. The transform length, n = length(x) or shape(x)(dim), can be any integer >0, but the method is most efficient when n is the product of small primes. x is assumed to be periodic in n+1. See also the inverse transform, ffti.
- ffti(x;dim) returns the inverse of the Fourier transform `fft`. For x real or complex, ffti(fft(x)) = x * n for x one-dimensional, where n = length(x), and ffti(fft (x,dim),dim) = x * shape(x)(dim) for any x with dimensionality $\geq$ dim.

77.2 Detailed Documentation

Basis built-in functions `fft` and `ffti` are the interface to the SLATEC subroutines `cfftf`, `cfftb`, `rfftf` and `rfftb`. Data can be real or complex, and the length of the transforms *N* can be any number, but the method is most efficient when *N* is the product of small primes.

77.2.1 *Transforms of one–dimensional data*

For any `x` periodic in N+1,

```
ffti(fft(x))/N = x
```

When *x* is a complex vector of length *N*, here regarded as subscripted *j*=0,...,*N*-1, `fft (x)` returns

$$z_k = \sum_{j=0}^{N-1} x_j \exp\left(\frac{-2\pi i j k}{N}\right), \tag{77.1}$$

and the inverse `ffti(x)` differs only in the sign in the exponential. Here the designation "inverse" is arbitrary; either transform followed by the other returns the original values multiplied by *N*, i.e. `ffti(fft(x))/N = fft(ffti(x))/N = x`.

When *x* is a real vector of length *N*, regarded as subscripted *j*=0,...,*N*-1, `fft(x)` returns a real vector *z* of length *N*, defined as follows: Let *l* =*N*/2 for *N* even, and *l* =(*N*+1)/2 for *N* odd. The real parts (cosine coefficients) and imaginary parts (sine coefficients) of the complex transform are

$$c_k = \sum_{j=0}^{N-1} \left[x_j \cos\left(\frac{2\pi jk}{N}\right)\right] \tag{77.2}$$

, and

$$s_k = -\sum_{j=0}^{N-1} \left[x_j \sin\left(\frac{2\pi jk}{N}\right)\right] \tag{77.3}$$

.

These Fourier coefficients are returned as `fft` = *z* = *c*0, *c*1, *s*1, ..., *c*l-1, *s*l-1, *c*l for *N* even, and `fft` = *z* = *c*0, *c*1, *s*1, ..., *c*l-1, *s*l-1 for *N* odd.

These *N* values include all the distinct coefficients.

The inverse transform *y* =`ffti`(*z*) returns *y* = *Nx*,

$$y_j = z_0 + (-1)^j z_{N-1} + \sum_{k=1}^{l-1} 2\left[z_{2k-1}\cos\left(\frac{2\pi jk}{N}\right) - z_{2k}\sin\left(\frac{2\pi jk}{N}\right)\right] \tag{77.4}$$

for *N* even, *j*=0,...,*N*-1, For *N* odd, the term with the factor (-1)*j* does not arise.

77.2.2 *Transforms of multi–dimensional data*

If *x* has dimensionality at least *n*, `fft` (*x*,*n*) performs a transform over the *n*th subscript, for all values of the other subscripts. For example, if *x* is two–dimensional, *z* = `fft(fft` (*x*,1),2) is its transform, and *x* = `ffti` (`ffti`(*z*,1),2)/`length`(*z*) is the inverse transform.

File `convolve` in public library `basis` contains examples of one– and two–dimensional smoothing and of solving Poisson's equation.

CHAPTER

SEVENTYEIGHT

FIT: Polynomial Fitting

The FIT package consists of two functions that implement polynomial fitting and a subsequent evalation of that fit:

- fit(x,y,n) returns an array c(0:n) of coefficients of the n'th degree polynomial which best fits the points y as a function of x in a least square's sense. The element c(i) is the coeficient of x**i.
- fitvalue(xx;c) returns the values of the polynomial described by `c` at the points `xx`. The polynomial coeficients c are as returned by `fit`, and default to the set returned by the last call to that function.

The routine fit causes these variables in the fit package to be set.

```
**** Fit:
# Results of calling fit
fitn integer /-1/
   # degree of the polynomial
fitc(0:fitn) _real
   # coefficients
```

-

CHAPTER

SEVENTYNINE

The History Package h2

79.1 A Facility for Iterative Programs

Specifying package h2 results in a package being loaded whose name is `hst`; this package is a second-generation version of hst which relies on the pfb package for its implementation.

Programs which contain an iterative step, such as a time step, often need to collect the values of variables after some or all of the iterative steps. This package assumes that there is an integer variable which is incremented after each iterative step, called the cycle-counter, and possibly an independent variable, often representing time, which increases monotonically with the cycle-counter. The package allows the user to periodically collect values of arbitrary expressions, using a variety of mechanisms to select the frequency at which the values are collected. Each value of a given quantity is called a *generation*, while the entire collection is called its *history*. For example, if a scalar quantity x is collected 20 times, then the history of x is an array of 20 values, each of which is referred to as a generation of x.

This package allows many sets of quantities to be collected with differing conditions governing the selection of generations, and allows different cycle-counters and independent variables for each collection.

79.2 Tags

The history mechanism is based on the concept of a history *tag*. Associated with each history tag are:

- A list of items whose history is to be collected.
- The place the history will be collected (file or memory).
- The name of the scalar real variable, if any, which is to be used as the independent variable.
- The name of the scalar integer variable which is to be used as the cycle-counter.
- Conditions determining when a generation is to be collected.

- A numerical priority that controls the order in which tags will be collected, if they otherwise are collected at the same time or cycle.

User commands can be used to:

- Declare a new tag.
- Add an item to the tag.
- Set the name under which an item will be stored.
- Change the conditions determining when a generation is to be collected.
- Change the priority associated with the tag.

The routine *hstory* is then called after each cycle of the iterative procedure. The conditions governing the collection of a tag can be changed at any time. Once the first generation of a given tag has been collected, items can not be added to it and the names under which the items are stored cannot be changed. New tags may be created at any time.

79.2.1 Definitions

1. A tag contains *items* whose history is to be collected. An item is a string that defines any Basis expression. They must be no longer than 72 characters in length. (If something more complicated is needed, make the item the value of a user-defined function which returns the desired value).

2. The condition determining when a generation is to be collected consists of a set of numerical and logical conditions as follows:

   - One or more of the following conditions on the cycle-counter or independent variable:

     (a) Start, stop, and increment conditions, or,

     (b) A list of values at which to collect;

     and

   - A logical-valued expression in Basis Language.

   An item is collected if it meets one of the conditions on its cycle-counter or independent variable **and** the logical expression evaluates to true. The default is to use a single condition on the cycle-counter: starting now, never stopping, and collecting every cycle, with logical condition “true”.

3. An *action* can be associated with a tag. When it is time to collect a generation of the tag, the associated action is executed before any data is collected.

4. The numerical *priority* associated with a tag affects the order in which tags are collected, if they otherwise would be collected at the same cycle or time. Priority is a floating point value. Zero is the default value. If two tags have different priority, the tag with the higher numerical priority will be collected first at a given cycle. If two tags have equal priority, the first tag defined is the first one collected at a given cycle.

79.3 Installation and Use

To add the `hst` package to your program, add the packages `h2` to your Dirlist. This will automatically get everything you need. If you are not using `mmm` we suggest you use it to produce a sample makefile from which you can extract the correct loading incantations for a given site. These vary so much from site to site that we will not attempt to list them here.

79.4 User Interface

This section describes the command interface used by the user of a program containing the `hst` package. A later section describes the subroutine interface, which may be used by either a user or an author.

A note on syntax: the user interface is implemented using the Basis command syntax and macros. Unless otherwise noted, the arguments to the history commands are space or comma delimited, macros are not expanded in collecting the arguments, and string-valued arguments need not be quoted unless they contain spaces. Spaces and commas inside parentheses do not count as delimiters. The effect is that you get what you want if you type expressions in a natural way, but without extraneous spaces unless inside parentheses. The macros inside items are expanded when it is time to evaluate the item.

1. Declare a new time history tag.

   ```
   newtag <tag> [filename]
   ```

   - This command declares a new history tag. It is not necessary to use this command if the tag is to use the default device; the other commands that require a tag name, such as `collect`, will create the tag for you. Macros are expanded in collecting the arguments, which are string-valued. If filename is omitted, the tag is kept in the default file. The default file's name is taken from the package variable `hstdev`. If filename is blank, the tag is kept in memory. The initial value of `hstdev` is blank, so unless `hstdev` is changed, the command `newtag <tag>` keeps the history in memory. The routine `hstsdev` can be used to change `hstdev`.
   - The name of the independent variable is taken from the history package variable `hsttime`. The routine `hststim` can be used to change `hsttime`. If `hsttime` is blank, there is no independent variable for this tag.

- The cycle-counter variable's name is taken from the history package variable `hstcycle`. The routine `hstscyc` can be used to change `hstcycle`

Every tag contains the following items initially:

- An item corresponding to the cycle-counter.
- An item corresponding to the independent variable, if there is one.

2. Add an item to the tag. There are two forms of the command for adding items to a tag; the second form is simply a short-hand way of listing many items that have the same subscript or function arguments.

```
items <tag> <itemlist>
items <tag> [elements <elementlist> of] <variablelist>
itemsv <tag> <itemlist>
itemsv <tag> [elements <elementlist> of] <variablelist>
```

An `<itemlist>` is simply a list of the items to be collected, space or comma delimited. Each item is a string specifying the expression to be collected. The itemsv form must be used if the item represents a quantity whose shape or type may vary over time.

An item may terminate with an at-sign (@) followed by a name; if it does, the name is used as the name of the history. Otherwise, the name under which the item will be stored in memory or a file is set to `<tag> <item>`. If the item is to be stored in a file, the name may be adjusted to make it a legal name for the database used.

When you have one or more variables you wish to collect at a set of subscripts, the second form allows you to list the set of subscripts in the `<elementlist>` and the names of the variables in the `<variablelist>`. Every combination of variables and elements becomes an item added to the tag. The `<elementlist>` subscripts should include the parentheses. Do not use the at-sign notation with this form.

This command must be executed before the first collection of the tag occurs.

3. Associate an action to the tag.

An expression can be associated to a tag; this expression will be executed when it is time to collect the tag but before collecting the next generation. This expression is called the tag's *action*. The usual purpose of an action is to calculate the values of some variables that belong to the tag. It is only executed after determining that the conditions on collecting the tag have been met.

```
tagaction <tag> <action>
```

where `action` is a string containing any Basis Language expression, sets the action for `<tag>`. Omitting `action` deletes the tag's action. The second argument includes everything up to the next semicolon or the end of the line, with no macro expansion. When the action is executed, macros will be expanded. To include a semicolon in the action, enclose the action in quotes.

4. Change the tag priority.

   ```
   tagpriority <tag> <priority>
   ```

   The default for `priority` is 0.0. Higher values indicate higher priority.

5. Change the conditions determining when a generation is to be collected.

   When a tag is created, its condition is initialized to cycle-counter condition start = now, stop = never, step = 1, logical condition `true`. This condition can be changed or added to with the `collect` and `andcollect` commands. The collect command replaces all existing conditions on the cycle-counter or independent variable for a given tag. The andcollect command is identical to the collect command, except that it adds the conditions to the existing set.

   There are three forms of these commands: with the tag name and one real or integer argument; with the tag name and sets of three scalar integer or real arguments; with the tag name and one scalar string argument. Macros are expanded in all arguments except the tag name, and all arguments are expressions, not strings.

   - Specifying a list of specific values.

     ```
     collect <tag> <list>
     andcollect <tag> <list>
     ```

     The collect command declares a list of values of the cycle-counter or independent variable at which the generations of `<tag>` are to be collected. `<list>` is either a vector of real values of the independent variable at which to collect or a vector of integer values of cycle numbers at which to collect. The values need not be sorted. If the appropriate variable passes over more than one value in the list in a single cycle, only one sample is collected. The collect command replaces all existing conditions on the cycle-counter or independent variable for a given tag. The andcollect command is identical to the collect command, except that it adds the list to the existing set.

   - Specifying start, stop, and interval values.

     ```
     collect <tag> <start> <stop> <step> ...
     andcollect <tag> <start> <stop> <step> ...
     ```

     This command specifies start, stop, and increment values governing the collection of generations of the `<tag>`. The type of the values `<start>`, `<step>`, `<stop>` determines which kind of limits these are, cycle or independent variable. If `<start>` or `<stop>` is real, and `<step>` is integer, the increment used is(`<stop>`- `<start>`)/(`<step>`-1). There may be as many sets of three values as desired. The collect command replaces all existing conditions on the cycle-counter or independent variable for a given tag. The andcollect command is identical to the collect command, except that it adds the conditions to the existing set.

   - Specifying a logical condition.

     ```
     collect <tag> "<condition>"
     andcollect <tag> "<condition>"
     ```

- A string value for the second argument sets the logical condition under which the generation of `<tag>` is to be collected. (Note that the quotation marks are usually required since the arguments to the collect command are expressions.) `"<condition>"` must be a string which can be evaluated to yield a logical value in the Basis Language. The `andcollect` command sets the logical condition to the string `(present)&(<condition>)` where `present` is the current value of the condition.

6. Collect history. There are six functions available:

```
call hstall
call hstalll
call hstallc
call history("<tag>")
call historyl("<tag>")
call historyc("<tag>")
```

   The routines `hstall`, `hstalll`, and `hstallc` each call `hstory`, `hstoryl`, and `hstoryc`, respectively, for all tags. The argument to the latter routines is the name of a specific tag.

   The essential routine is `hstory`. Routine `hstoryl` is used at the end of a problem, and `hstoryc` can be be used to check items before beginning a run.

   - `hstory` is meant to be called after every increment to the cycle counter of a tag is completed. It decides whether it is time to collect a generation of the tag, and if so, executes any action associated with the tag, collects and stores the data for each item, and resets the conditions for the next generation to be collected.
   - `hstoryl` is a variant of `hstory` for collecting a "last point". It collects a generation of every tag for which there is a pending collection value, even though that time has not yet been reached, as long as the logical condition is met.
   - `hstoryc` attempts to check the items belonging to the tag for validity. It does this by attempting to evaluate the item. This can fail even when the item is in fact valid. Some examples of this are: if an array is not currently allocated space but will be by the time the item is collected; if an item involves an arithmetical calculation which is not valid now but will be when the tag is collected.

7. Displaying the status of tags.

```
call hstallp
call hstprint("<tag>")
```

   Routine `hstallp` calls `hstprint` with the name of each tag in turn.

   `hstprint` prints a report of the status of each tag to the terminal.

79.5 Dumping and Restarting

All the variables, including any time histories generated during a run, that need to be preserved over a dump/restart, have the attribute “dump”. Thus, to dump the history package you need only ask the `pfb` package to create a file and then dump all variables with attribute “dump” to it. A typical method is:

```
integer fileid, pfbopen
fileid = pfbopen("mydump","w")
call pfbsave("all",fileid)
call pfbasave("dump",fileid)
call pfbclose(fileid)
```

After restoring from a file containing this package, you *must* call the routine `pfbhst`.

```
call hstrest
```

79.6 History Arrays

There are two kinds of history items, those created with the ”items” command, which are of fixed shape and type, and those created with the ”itemsv” command, which may vary in shape or type.

For normal ”items”, the first time a history is collected in memory, the history array is created as the first generation with an extra dimension added to it. Any leading dimensions of size 1 are squeezed out. Thus, if h is the name of the history, and x is the item, the result is:

```
hst chameleon h = squeeze(x)
call rtadddim("hst.h")
```

As subsequent generations are collected, the new value of the history will be the result of the Basis expression

```
hst.h:=x
```

which must be a legal expression. This means that if x is always a scalar, then h(i) is the i’th generation ; likewise h(,,i) is the i’th generation if x is a two-dimensional array.

For items declared with the ”itemsv” command, the semantics of subsequent collection are:

```
hst.h:=[hst.h,x]
```

Note that therefore changes of size or shape will obliterate the distinctions between the generations unless the user also collects auxiliary information to use in decoding the resulting history. Example: suppose y is a one-dimensional array which changes its length. Then besides collecting y, we should collect length(y) so we can calculate where the generations of y begin in the history, which will be a one-dimensional array rather than a two-dimensional array.

79.7 Deciding When To Collect

The conditions the user can set can either be on the independent variable or on the cycle-counter. The user can specify a set of such conditions for each tag; the tag will be collected whenever any one of the conditions is satisfied (provided the logical condition is satisfied too).

We can view the start-stop-step form of a condition as specifying a list, so the problem of determining when to collect a generation can be phrased in terms of the list-type condition. The value of the cycle-counter or independent variable we will call the “current value”. We call the value at which the next collection may occur the “pending value”.

When a tag is created, the cycle at which the tag was last collected is set to minus infinity. The tag is marked “active”.

When a collect command is executed, a value for the condition called the pending value is set to the smallest element in the list. The tag is marked “active”.

When the routine hstory is called, no action is taken if the tag is inactive. If a tag is active, a tag is collected if, for one of the tag’s conditions, the current value is greater than or equal to the pending value and the logical condition is true, and the current cycle-counter is larger than it was the last time this tag was collected.

When a tag is collected, the cycle-last-collected is set to the current value of the cycle-counter. The pending value is recalculated as follows, for each condition belonging to the tag. (The pending value is also recalculated if it is time to collect the tag but the logical condition is false). If the current value is smaller than the maximum value in the list, the pending value is set to the smallest element in the list which is strictly larger than the current value. If there is no such element, the condition is removed. If there are no conditions remaining, the tag is marked “inactive”.

79.8 Examples

79.8.1 A simple tag kept in memory

```
hsttime="time"
hstcycle="ncyc"
items t1 a,b
collect t1 0 100 10
run  # run the program, which calls hstall
plot t1a,t1time
plot t1b,t1time
```

This example assumes that a and b are scalar quantities, so that `t1a`, `t1b`, and `t1time` are vectors. The collect statement sets this quantity to be collected every ten cycles up to and including cycle 100.

79.8.2 How to deal with fancy names

The history package creates history names which may be long or clumsy. You can use the `@name` option in an `items` command to make the history have a simpler name. If you do not, you can make it easier when accessing the variable later in several different ways. For example:

```
define y timehistr_3_2_
indirect z = "timehistr_3_2_"
function w(;k)
default(k)=1:length(timehistr_3_2_)
return(timehstr_3_2_(k))
endf
```

makes `y`, `z`, and `w` all easy ways to get at `timehistr_3_2_`.

79.8.3 A tag kept in a file, collected subject to a condition

```
newtag blue junkfile
items blue x,y,z(20)
items blue elements (4) (7) (10) of yw
items blue elements (4,5:12) of www
collect blue "energy > 10"
collect blue 0. 3. .2
```

79.8.4 Using the macro processor

Since macros are not normally expanded in an items command, we need to use a macro which expands into a list of quoted items, and then precede the macro name with a caret escape. For example:

```
mdef myzones=  "(3,4)" "(5,10)" "(6,10)" mend
items green elements ^myzones of a,b,c,d
items green elements ^myzones of e,f,g
collect green 0 10000 20
run
plot 'greena(3,4)','greenb(3,4)'
```

79.8.5 A tag collected at a list of times

```
collect charged [1.,2.,3.,4.4]
items charged a@ahist,b@bhist,c@chist
```

In this example, the history of a is collected as ahist, that of b as bhist, and that of c as chist, rather than using the default names chargedhista, etc.

79.8.6 A tag collected at log intervals

```
newtag neutral file
neutral d,e,f
#collect neutral at 1.e-5, 1.e-4, ...., 0., 10.
collect neutral 10.**iota(-5,1)
```

79.8.7 Function items

```
# mass and volume are code variables dimensioned (k,l).
function density(i,j)
default(i)=1:k
default(j)=1:l
return mass(i,j)/volume(i,j)
endf
items yellow density(2,3)@den23
items yellow density
collect yellow [10,20,30,40,44]
```

At cycles 10, 20, 30, 40, and 44, the following quantities will be collected:

```
density(2,3)    #history named 'den23'
density(1:k,1:l)
```

79.8.8 A traveling probe

Imagine the program contains one-dimensional arrays x and y, and we want to track the values of x and y at the point at which y is a maximum.

```
real x1, y1
items probe x1, y1
collect probe 0., 10., .1
tagaction probe "global real x1=x(mxx(y)), y1 = y(mxx(y))"
```

When it is time to collect probe, the action is executed so that x1 and y1 have the desired values.

Here is another way to accomplish the same thing:

```
items probe2 x(mxx(y)), y(mxx(y))
collect probe2 0., 10., 1.
```

For further examples see the test routine, test.hst. It is located in the `hst` library.

CHAPTER

EIGHTY

PFB Package

80.1 Summary

PFB is a Basis package (Portable-Files-from-Basis) which adds an interpreter interface built on top of a Fortran interface to portable database files. The PFB package can be easily added to any Basis program (See 80.8.) On installations where PDB is present, the program basis usually includes the PFB package. PFB can be used with or without the PDBSAV package. Currently, the only database format available is PDB.

open *filelist*

openg *filelist*

ls [-aflrs] *varlist*

close *fileid*

record *[number]*

jt *when*

create *filename*

write *varlist*

writeas *expression,name*

writef *varlist*

restore *filelist*

80.2 Reading Files

80.2.1 File Numbers

As each file is opened for read or write, it is given a number. The list of files and their numbers can be seen using the **ls -a** command. Once a particular named file is opened, it always retains the same number even if it is closed and reopened later.

The current file open for read is the last one specified in an **open** command. The current file open for write is the last one specified in a **create** command. These commands can be used to switch attention between several files open for reading and writing.

80.2.2 Opening and Closing Files

open

Calling Sequence

```
open filelist

open filenumber
```

Description

`open` opens each file for reading. The list can be comma or space separated. The names of the files need not be quoted. If you wish to open a file whose name is the result of a Basis expression, preceed the expression with a caret. If an open command is given on a file open for write, it is first closed.

The Basis path is searched for the file if it is not in the current directory. If more than one file is specified, they are opened in the order given and the last one becomes the current read file.

If a file is already open, the `open` command can be used to make it the current read file. In this case, the file number can be given in lieu of the file name. The variable `pfbofam` governs whether or not a family of files is opened as a whole.

openg

Calling Sequence

```
openg filelist
```

Description

`openg` is a command for connecting together history files which pfb either does not recognize or which it has not opened as a unit due to the `pfbofam` option being no.

The arguments can be file names or the fileid numbers of previously accessed files.

Each file is opened normally, observing the convention to open subsequent family members or not depending on the status of pfbofam. Each successive file is “glued” to the first, and then closed, so at the end only one file sequence is open, the first, which contains records that span the entire set.

The files should be given in order of increasing sequence number. Example: a user has a family file00, file01, file02, file03, but file02 has been lost. With `pfbofam=yes`, we do:

```
openg file00 file03
```

Please note that if you close a glued sequence the gluing is lost. In the above example about the missing file02, closing file00 and then doing open file00 would result in only opening files file00 and file01 as a unit.

The process of gluing is carried out by routine

```
pfbglue(fileid1, fileid2)
```

which is Fortran or Basis callable. The two arguments are the fileids (not the names) of the two pieces to be glued. The sequence represented by fileid2 is "glued" to that of fileid1 and then fileid2 is closed.

Gluing means: The last record kept from the first sequence is the last one whose cycle number is strictly less than the first cycle number from the second sequence. It is an error if there is no such record. Given the set of files f00,f01,f02,f03, the following two lines lead to equivalent sequences open under the name "f00".

```
pfbofam=yes; open f00
pfbofam=no;  openg f00 f01 f02 f03
```

Other than the check on the cycle number, no attempt is made to see if the operation of gluing makes sense. In particular, the only accessible variables are those occurring in the first file, and it is assumed they occur in the later files with the same size and type.

It is not yet ok to open together files with different representations, such as part of a family from a workstation with part from a Cray.

close

Calling Sequence

```
close [fileid]
```

Description

Closes a file so that its variables are no longer visible to Basis. Files not otherwise closed are closed when the program ends. If fileid is not given, the file currently open for write is closed, if there is one. Otherwise the file currently open for read is closed.

80.2.3 Disambiguating Variables with Identical Names

If an open file contains a variable with a given name, say *foo*, the question arises of how to refer to this variable in preference to another variable in the program which has opened the file. The

variable in the file is considered a variable in the pfb package, and as such its "full" name is *pfb.foo*. If you wish to make file variables have precedence over compiled code variables you can give the pfb package a higher precedence with the command

```
package pfb
```

User-created variables have a higher priority than file variables unless you set `usrfirst = false`.

80.2.4 Listing Files and Their Contents

ls

Calling Sequence

```
ls [-aflrs] [varlist] [-x ls_options]
```

Description

The `ls` command can list information about files opened or created, and about variables and time history records in the current file open for read. The options can be given separately or together (`ls -ar` or `ls -a -r`, for example).

The `-f` option causes `ls` to print information about the open files. Files open for read will be preceded by ">>", those open for write will be preceded by "<<". The current file open for read, and the current file open for write, will be marked with a plus sign.

The `-a` option causes all files that have been accessed to be listed, even if they are not currently open.

The `-r` option will list information about the current family of files and the times and cycle numbers of therecords in each. See the discussion of time history files, below.

`ls` will describe the variables in the current database file. The *varlist* can be a space or comma delimited list of names or keywords. Each name given will be described. If no argument is given, all entries in the given file will be described. `ls` can list information about the chosen variables in two forms, short and long. The short list lists only the names of the variables in the file and is the default. The default can be permanently changed by setting the control variable `ls = yes` or `no`, `yes` meaning a short list. (See CONTROL VARIABLES.) The default form of the listing can be overridden with `ls -l` or `ls -s` for long and short forms respectively. `ls` *foo.** will list only those variables in the file whose package designator is *foo*. Macro texts have a package designator of "macro". Functions have a package designator of "funct". `ls` *foo** will list those variables whose names start with *foo*.

When a `ls` command is issued when no file is open for read, the files of the current directory are listed instead. The `-x` option can be used to add directory listing to ordinary variable listing requests.

If no file is open for read, the arguments to the `ls` command, if any, are passed on to a call to the standard Unix ls. (The actual command executed is in control variable `pfblsopt`. The default value is “pwd;/bin/ls” to which any arguments are appended. Remember that the command is executed by the Bourne shell.)

If a file is open for read, you may add a `-x` option, followed by other arguments, to be passed on in the same fashion, at the end of an ordinary ls command (in this case, if `-x` is the first option, no list of file variables is done).

80.2.5 Accessing Variables

All the variables in the current read file will be known to the interpreter. Each variable in the file has an official name by which it is known in the file. If that name contains an at-sign (@), as it does for any file written by the `write` command, or `pfbsave`, the part before the at-sign is the “short name” and the part after it is the “package name”. A short name can be used to access a variable from the interpreter; if this name is ambiguous the first such variable encountered in the file is returned. The package name is the name of the Basis package to which the variable belonged, or a keyword “macro”, “funct”, “history”, “open”, “hidden”, or “record”, used to indicate that the variable has contents with a special meaning. If no at-sign occurs in the variable name, both the short and long names are equal to this name, and the package name is blank. (The only way to write a item whose full name does not have at at-sign in it is to use the `writeas` command, below, and end the target name with an at-sign.)

If a name is specified that contains a percent sign (%) followed by an integer, and that integer corresponds to the number of a file which is open for read, it is interpreted to mean that the name up to the percent sign is to be read from the file of the corresponding number. The current file does not change. Note that any name containing a % must be surrounded by single quotes.

Example

For example, if files *abc* and *def* both contain a variable named *x*, then their difference could be printed with:

```
open abc, def; 'x%2' - 'x%1'
```

The `'x%2'` could just be a plain `x`, since at that point *def* is the current file.

See also ACTIONS WHEN OPENING A FILE, below.

80.3 Writing Files

80.3.1 Creating or appending to PDB files

create and append

Calling Sequence

```
create filelist
append filelist
```

Description

Creates each file named in the list. If a file exists but is not currently open for write, it is destroyed. The control variable `pfbask` controls whether or not the user is given a chance to object to this. (See CONTROL VARIABLES). The `append` command can be used to open a file and then writing more information into it..

If a file is already open for write, the create command can be used to make it the current write file. In this case, the file number can be given in lieu of the file name.

80.3.2 Writing Information to Files

write

Calling Sequence

```
write list
```

Description

Given a *list* of comma-delimited list of items, each of which is a variable name or basis expression, writes their contents into the current output file. The keyword “functions” causes all user defined functions to be written. The keyword “macros” causes all macro definitions to be written. The keyword “variables” causes all user-created variables in the global database to be written. The keyword “all” writes macros, functions, and variables.

The name used to store the value in the file will depend on the nature of the item to be written. For the name of a variable, it will be *name@pkg*, where *pkg* is the package to which the variable belongs. For an expression, the name will be *e@value*, where *e* is derived from the text of the expression by replacing all non-alphanumeric characters by underscores and truncating characters in excess of 24. It is the users responsibility to avoid name collisions between different items.

If an item is simply the name of a macro or function, the macro or function it must accept being called with no arguments. The name used to store the result is *name@value*. (See `writef`, below, for storing the function or macro text.)

writef

Calling Sequence

```
writef list
```

Description

The `writef` command works the same as `write` except in the case of an item which is the name of a function or macro, in which case the function or macro definition is stored, not the value obtained by calling it with no arguments. The name used to store it in the file is *name@funct* or *name@macro*.

writeas

Calling Sequence

```
writeas expression name
```

Description

Writes a variable into the file with file name *name* with a value equal to *expression*. If *name* does not contain an ampersand, "@value" will be appended. If *name* ends in an ampersand, the name used will be *name* less the final @.

80.4 Restoring From A FIle

80.4.1 The restore command

While the normal read procedures can access the data in a file, they do not bring it into memory as a permanent variable. The `restore` command is used to bring in variables and store them in appropriate places in the receiving program. The most common reason for doing this is as part of a restart procedure that allows continuing a long calculation whose state was saved. Most commonly this is done using a combination of `write` commands with the attribute server routine `pfbasav`e.

restore

Calling Sequence

```
restore filelist
```

Description

Opens each file in the list, restores the variables in it into the code, as described below, and closes it. *Filelist* can be comma or space separated. The names of the files need not be quoted. If you wish to restore from a file whose name is the result of a Basis expression, precede the expression with a caret. If you wish to restore selected items from a file, see pfbrest.

Restoring is the act of putting back into memory the values in a file written by PFB. This is done according to the following set of rules. Each "item" in the file is treated in the order written.

- A macro is restored if it is NOT currently defined.
- A Basis function is restored if it is NOT currently defined.
- A structure is not restored.
- A history variable is not restored.
- If the package name is `value` the variable is not restored. Such variables result from `writeas`, above.
- For each other item in the file the following protocol is followed. If the package name is not that of a package in this program, it is changed to "global". Then:

1. If a variable name corresponding to the package name and short name exists, the values are restored to it. If the variable is dynamic, memory is allocated for it using the dimensions of the item in the file. Any existing contents are lost.
2. If there is no corresponding program variable, it is created and the values restored to it. Its "original shape" string is set to reflect the current shape.
3. An existing chameleon variable also has its shape set to the new size.

For an existing variable, if there is not a perfect match between file and variable in terms of size and type, the file variable is read into memory and an assignment is attempted (as if executing `codevar = filevar`). If the program variable is statically allocated, an assignment is attempted. If it is dynamically allocated, it is allocated at the correct number of elements with its current type.

Cautions

- A dynamic variable with the right type but a completely different shape can chameleon itself to the new shape under these rules.
- There is no checking against the compiled dimension string in these cases. A call to baschange after restoring may be in order if you aren't playing straight with PFB.
- Dynamic, limited variables which are saved and then restored will be at the limited size, which may make them inconsistent with their dimensioning string.
- Restoring into a different program than originally wrote the data is permitted but errors may occur due to conflicts in names, types, or shapes.

Following a `restore`, the function `pfbrerrs()` returns the number of errors due to these causes.

An individual item may be restored by using the functional interface `pfbrest`.

Example: Simple Dump/Restart Using Basis

```
    create mydump
    write all
    call pfbasave("dump")
    close
# on a later date ...
    restore mydump
```

Example: More Sophisticated Dump/Restart From Fortran

Here is a typical invocation from Fortran, saving all user functions, macros, and variables along with all variables that have the attribute `changed`, `keep`, or `dump`. The calls to `pfbalist` are to ensure that variables to which the user has given these attributes at run-time are able to be given the attribute after restore by `pfbaset`.

```
    subroutine dumper(dumpname)
    character*(*) dumpname
    integer fileid, pfbopen, basisexe, status
    external pfbopen,basisexe
    logical isthere
    character*300 basiscmd

    inquire(file=dumpname,exist=isthere)
    if(isthere) then
          basiscmd="/bin/rm "//dumpname
          status = basisexe(basiscmd)
    endif

    fileid = pfbopen(dumpname,"w")
    call pfbalist("v_changed","changed")
    call pfbalist("v_keep","keep")
    call pfbalist("v_dump","dump")

    call pfbsave("all",fileid)
    call pfbasave("dump|keep|changed",fileid)

    call pfbclose(fileid)
    return
    end

    subroutine restart(dumpname)
    integer fileid, pfbopen
    external pfbopen
    character*(*) dumpname
    integer space
```

```
    fileid = pfbopen(dumpname,"r")
    call pfbrest(fileid," ")
    call pfbclose(fileid)
# needed only if h2 package used
    call hstrest
# re-establish attributes keep, dump, changed
    call pfbaset("global.v_keep")
    call pfbaset("global.v_dump")
    call pfbaset("global.v_changed")
    return
    end
```

80.4.2 pfbrest(fileid,name)

The `restore` command is actually implemented via the routine `pfbrest`, which may be called directly if you wish to restore just a particular item from a file.

80.4.3 pfbrs

`pfbrs(name;fileid)` calls `pfbrest(fileid,name)`. This allows you to restore one item from the current file without explicitly referring to the fileid since the second argument defaults to it.

80.5 Time Histories

80.5.1 Beginning and Ending Records

PFB can write variables periodically to a file so that in the file they appear to have an extra final dimension representing time. The normal write functions are used to write history variables. To begin time history output, one first calls `pfbbegr`. At the end of a set of writes for that time, call `pfbendr`.

History files written by the old dmi2pdb interface can be read by PFB. The package name *record* is used for each data member. Data members whose names contain a period cannot be accessed. The names of the structures and certain auxillary variables are also hidden from the user.

History files written by the POP-to-PDB translator can be opened by PFB correctly. To get a correct time catalog, first set `pfbdtime = "time@history"`.

80.5.2 File Families

The integer function `pfbfam(fileid)` looks at the current file being written associated with *fileid*, and if it contains more than the number of words in the control variable *pfbmax*, it closes that file, opens the next file in the sequence, and returns the new fileid. Otherwise it simply returns *fileid*.

80.5.3 Reading History Values

When reading history variables, the user may specify all dimensions, but if the user does not explicitly supply the final (time) index, it defaults to the current value determined by the `record` command or the `jt` command, given below.

When supplying the final index, the user may give an integer record value or a real value representing a time. The latter will be converted to an integer representing the record whose time is nearest the given time. In this case the current record number is not affected.

record

Calling Sequence

```
record [n]
```

Description

`record` sets the current record number to *n*. If no record number is given, the current record number is printed.

jt (jump to time)

Calling Sequence

```
jt t
jt n
```

Description

`jt` sets the current record to the given time or cycle n. The type of the argument determines whether it is interpreted as a time or as a cycle number. See also the functions `pfbjc` and `pfbjt`.

History File Details

PFB treats a variable in a file as a history variable if and only if the package name of the variable is *history* or *record*. , A file is treated as a possible familied file for reading if and only if it contains

at least one history variable. The control variable *pfbofam* can be set to *no* in order to open only one member of a family.

Caution

When a family is open, do not open explicitly any other member of the family except the first one.

80.6 Actions When Opening a File

`writeas` can be used to store a scalar or array of type character into a file under a name *something@open*. If such a file is later opened, the entire text of each item whose name ends in *@open*, treated as one long character string, will be parsed as Basis Language when the file is opened. (An `open` command done for the purposes of switching from one open file to the other does not trigger this, just the initial open.)

This behavior can be suppressed by setting the variable `pfbact = no`.

80.7 Control Variables

A number of variables are available to control the detailed behavior of the PFB package.

- **pfbdebug** controls the debugging output of the pfb package. The amount of extra detail increases as you increase the value. (default 0)
- **pfbask** controls what happens to an existing file when asked to create a file by that name. Think hard before setting yes in a batch job. (default:no)
- **pfbls** controls whether or not the default listing mode is short (default, yes)
- **pfbmax** is the number of words a file can contain before `pfbfam` will family it. (default 250000 words)
- **pfbcycle** is the name of the variable to use for cataloging cycles when creating a history file.. If blank, the record number is used. Default: blank
- **pfbtime** is the name of the variable to use for cataloging time when creating a history file. If blank, the floating point record number is used as time. Default:blank
- **pfbofam** if set to `no` prevents more than one family member from being opened. record and `jt` still work within the one file. Default: yes.
- **pfbhide** if set to no lists variables in old-stylerecord files that you normally shouldn't see.
- **pfbhide** if yes, don't show dmi2pdb superstructure in dap-style old history files (yes)

- **pfbdcyc** informs PFB of name of cycle variable in non-standard file; set it before opening file (blank)
- **pfbdtime** informs PFB of name of time variable in non-standard file; set it before opening (blank)
- **pfbact** – on open for read, parse contents of variables named *@open? (yes)

80.8 Installation and Use

To add the `pfb` package to your program, add the packages `pfb` to your Dirlist. This will automatically get everything you need.

80.9 Functional Interface

These routines are callable from Fortran or Basis, except for the builtin functions which can only be called from Basis. When a semicolon appears in the argument list, it indicates that the following arguments are optional from Basis.

80.9.1 File manipulation

```
pfbopen(name:string;access:string) integer function
   # returns fileid
pfbopend(fileid:integer) subroutine
   # parse contents of each variable named *@open in file.
pfbclose(;fileid:integer) subroutine
   # close the file; defaults to write file, if any, else read file
pfbfile(fileid:integer) builtin [1]
   # return the name of the file given fileid
pfbfile(fileid1:integer,fileid2:integer) subroutine
   # glue family connected to fileid2 to fileid1, closing fileid2.
```

80.9.2 Writing

These routines assume *fileid* is the file id (returned by `pfbopen`) of a file open for write.

```
pfbsave(name:string;fileid:integer) subroutine
   # save item to file; can invoke with write macro
   # fileid defaults to file id of current write file
pfbsavee(expr, name:string, fileid:integer) builtin [2-3]
```

```
   # save expr as name
pfbasave(aexp:string;fileid:integer) subroutine
   # save things satisfying attribute expression aexp.
   # fileid defaults to file id of current write file
pfbalist(v:string, a:string) subroutine
   # create a list of variables satisfying a as variable v
   # first element of the list is a
```

80.9.3 Restoring

These routines assume *fileid* is the file id (returned by `pfbopen`) of a file open for read.

```
pfbrest(;fileid:integer,name:string) subroutine
   # restore from the file; restore just name if not blank
pfbrs(name:string;fileid:integer) subroutine
   # restore from the file; restore just name if not blank
pfbaset(v:string) subroutine
   #  Given v made by pfbalist, restore attribute to variables
   #  Call after doing the restore from the file
pfbrerrs() integer function
   # return number of errors in last call to pfbrest
pfbpad(jvar:integer, ndb:integer) integer function
   # User-replaceable function to pad variable being restored.
```

80.9.4 File catalog

These routines assume *fileid* is the file id (returned by `pfbopen`) of a file open for read.

```
pfbcount(;fileid:integer) integer function
   # number of names in current read file
pfblong(i:integer;fileid:integer) character*(NPDBN) function
   # return the long name of the i'th entry
pfbpack(i:integer;fileid:integer) character*(NPN) function
   # return the package name of the i'th entry
pfbname(i:integer;fileid:integer) character*(NPDBN) function
   # return the short name of the i'th entry
```

80.9.5 Time History

```
pfbjt(t:real;fileid:integer) integer function
   # return record number closest to given time
```

```
pfbjc(n:integer;fileid:integer) integer function
   # return record number closest to given cycle number
pfbbegr(;fileid:integer) subroutine
   # enter record mode
pfbendr(;fileid:integer) subroutine
   # leave record mode
pfbgrec(;fileid:integer) integer function
   # return current record number in file being written
pfbsrec(;irec:integer) subroutine
   # set record number for reading
pfbfam(;fileid:integer) integer function
   # return fileid or fileid of new family member of file, if full
pfbgoto(when) builtin [1]
   # set record number to last record before or equal to given time or cycle
```

80.9.6 Internal and Command Implementation

The following routines are not normally directly called by users from either Fortran or Basis.

```
file_access builtin [0-7]
   # Internal mechanism used by PFB package to access data in files
pfbopenl(namelist) builtin [0-100]
   # implements the open command
pfbopeng(namelist) builtin [0-100]
   # implements the openg command
pfbclosel(fileidlist) builtin [0-100]
   # implements the close command
pfbcreatel(namelist) builtin [0-100]
   # implements the create command
pfbappendl(namelist) builtin [0-100]
   # implements the append command
pfblist(stringlist) builtin [0-100]
   #ls [help|files|names|records]
pfblsrec(;fileid;unit) subroutine
   #ls records implementation
pfbwras(fileid,irec,name:string,typecode:integer,fwa:integer,
       ndim,ilow,ihi,icol)  subroutine
   # nitty-gritty output routine, do not try this at home
pfbrestl(namelist) builtin [0-100]
   # restore name1, name2, ...
pfbsavel(namelist) builtin [0-100]
   # write name1, name2, ...
pfbsavfl(namelist) builtin [0-100]
   # writef name1, name2, ...
```

CHAPTER

EIGHTYONE

SVD: Singular Value Decomposition

SVD supplies the routine `svd(a)`, which performs a singular value decomposition of the input matrix `a` and returns the results in variables in the svd package.

svd(x) calculates the singular value decomposition (svdu, svds, svdvt) such that x = svdu *! d *! svdvt, where svdu and svdvt are unitary, and d is a matrix whose first svdnm diagonal elements are svds. uses the appropriate lapack routines to return 64 bit precision answers

In Basis Language terms:

```
call svd(x)
real(8) d(svdm,svdn)
d = diag(svds)
svdu *! d *! svdvt => should be approximately x
```

The variables set by the call to `svd` are as follows. The precision of the real variables returned is 64 bit regardless of the precision of the input.

```
**** SVD:
svdm integer /0/
   #first dimension of most recent argument to svd
svdn integer /0/
   #second dimension of most recent argument to svd
svdnm integer
   # min (svdn, svdm)
svdu(svdm,svdm) _real
   # output u
svds(svdnm) _real
   # vector of singular values
svdvt(svdn,svdn) _real
   # v-transpose
svdinfo integer /0/
   # result code, 0 means ok
svdlw integer /0/
```

```
   # Work space used is at least 5*max(svdn,svdm)
   # svdlw holds the ideal amount suggested by Lapack
   # this is used on
   # the next call to svd with an identical problem size
svdwork(svdlw) _real
   # work space


 -
```

CHAPTER

EIGHTYTWO

TIM: Interrupt Timing

`tim` is a package which drives the 4 ms. interrupt p-counter sampling timer package. It is used in conjunction with the tally program. See file `tim.doc` for full instructions. `tim` is useful for finding out where your program (and Basis) is spending its time. `tim` works only on Cray machines. It is available as LIB file `tim` inside public library `basis`.

CHAPTER

EIGHTYTHREE

RNG: Random Number Generators

This package gives Basis codes an alternative random number generator, with support functions to query or set current seed values, and so forth.

83.1 The Mzran Suite

All random number generators like *ranf* suffer from a common problem in that if you plot successive values into two (or more) dimensions, the points fall on a series of lines (hyperplanes). It is possible to avoid these higher dimensional correlations by combining the output from two or more generators. Marsaglia and Zaman [1] showed several ways to do this recently; the following routines are based on their work.

All the routines in this group are available either to your compiled code, or from the Basis interpreter. The *mzran* generator is based on 32 bit arithmetic, and *uni32* returns *real(Size4)*.

83.1.1 Mzran

```
integer mzran,i
i = mzran()
```

Mzran returns random integers in $[-2^{31}, 2^{31} - 1]$. As noted by Marsaglia and Zaman, this routine provides flexibility to code developers in that you can easily write Fortran statement functions to rescale, translate, or mask its return value. *Mzran* is a compiled function in the Basis interpreter (not built-in.)

83.1.2 Uni32

```
real(Size4) uni32, rr
rr = uni32()
```

[1] *Some Portable Very-Long-Period Random Number Generators*, Computers in Physics, V8N1, Jan/Feb 1994, pp.117.

Uni32 calls *mzran*, then scales and translates the result to the interval $[2^{-32}, 1 - 2^{-24}]$. The minimum is the smallest positive IEEE 754 single precision value, and the maximum is the largest such value less than 1. Thus *uni32* can safely be relied upon to produce uniform deviates in the open interval (0,1). Like *ranf*, *uni32* is a Basis built-in function.

83.1.3 Setmzran

```
integer a,b,c,d
call setmzran(a,b,c,d)
```

The internal state of the *mzran* generator can be stored in four integer values, and *setmzran* changes its state to the four given arguments. *A*, *b*, *c*, and *d* can be any legal (32 bit) integers, not all zero. If all arguments are zero, the default values are reset.

83.1.4 Getmzran

```
integer a,b,c,d
call getmzran(a,b,c,d)
```

Getmzran retrieves the current state of the *mzran* RNG into the four integer arguments. If calling this subroutine from the Basis interpreter, be sure to pass the arguments by reference.

Part VII

MPPL Reference Manual

CHAPTER

EIGHTYFOUR

MPPL Reference Manual

84.1 A More Productive Programming Language

84.1.1 MPPL is a Fortran Preprocessor

MPPL ("More Productive Programming Language") allows programmers to write in a language that is more convenient and powerful than Fortran 77. MPPL then transforms statements written in the MPPL language into standard Fortran 77. This language is essentially an extension to Fortran 77 that provides free-form input and many structured constructs such as "while" and "for" loops. MPPL's macro preprocessor and file-inclusion facility encourage the creation of structured, easy-to-read programs that contain fewer labels. MPPL provides a more productive programming environment for Fortran 77 users on the Unix, Linux, AIX, IRIX, Solairs, HP-UX, Tru64 operating systems.

MPPL can be used independently as well as with Basis.

84.1.2 MPPL's Three Stages

During execution of MPPL, data flows through three ordered steps, or levels. The first level is the token processor; it reads the user's source code and divides it into "tokens", such as names, quoted strings, and punctuation marks. The second level is a macro preprocessor; it takes alphanumeric tokens that the user has defined as macros and replaces them with appropriate text. The third level is the statement processor; it reads tokens after they have been processed by the macro processor. Then the statement processor forms Fortran 77 output text, translating some higher-level programming constructs as it does so.

For most applications, a detailed understanding of the operation of MPPL is not required. The MPPL language is nearly upward-compatible with Fortran 77. The higher-level programming

constructs may be added to existing programs, or not, as the user chooses. MPPL does allow complicated macro definitions, but the basic usage is very simple:

```
define macroname expansion
```

causes subsequent appearances of the symbol `macroname` to be replaced by the rest of the line on which the define statement occurs.

The user can supply macro arguments like a function call, with the arguments in parenthesis and delimited by commas. The arguments are inserted into the expansion of the macro wherever the definition has a dollar sign followed by an argument number. Thus, an input of

```
define Pop   $1 = $1 - $2
Pop(k,n)
```

yields the statement

```
k = k - n
```

The macro processing facilities are similar to the Unix macro processor m4. The higher-level language facilities are inspired by the C language and the Unix utility Ratfor.

84.1.3 Read the Sample Programs First

Most users will find it suffices to read the next section to learn how to execute MPPL, and then read the MPPL program examples in section 84.7, referring as necessary to the syntax summaries in the Appendix. A more thorough understanding of the MPPL program can be postponed to the day when MPPL does something unexpected.

84.2 Execution

84.2.1 Availability

MPPL is available as `/usr/apps/basis/bin/mppl` at the Secure Computing Facility, the Open Computing Facility, and the A division networks. See http://basis.llnl.gov

84.2.2 Specifying Input and Output Files

To execute MPPL, the user specifies the files to be processed:

```
mppl file1 file2 ... filen
```

The names of the files that MPPL is to translate are delimited by spaces. Output is written to standard out.

A typical mppl-compile-load sequence is:

```
mppl mymacros mysource.m > myout.f
f77  myout.f -o xec
```

If MPPL is executed without a list of files it reads from standard input allowing it to be used as a filter.

Many mistakes in syntax will be caught by MPPL, such as missing `"endif"` statements, but compilation mistakes are possible since MPPL does not check all Fortran syntax.

84.2.3 Specifying Options

Options are entered first on the command line. Options and filenames may not be interspersed. If no files are given, or if a ‘’ alone is given, MPPL reads from stdin. All output is written to stdout, and error output to stderr.

`-`*N* Where *N* is a 1-5 digit integer, specifies the beginning value for MPPL-generated statement labels. The value should be chosen to prevent duplication of existing labels in your code. MPPL restarts the sequence in each subroutine. The default value of *N* is 23000.

`-b` Turn off the output of blank lines and comments. The default is to pass blank and comment lines to the output.

`-c` ***string*** Set the column 1 comment character to be any of the characters in *string* (up to three characters may be specified). The default value of this option is `cC*`.

`-C`*compiler* Specify the compiler to be used on the MPPL output. This sets the macro `COMPILER` to have the value *compiler*.

`-d` Convert literal character constants enclosed with quotation (`"`) characters to Fortran 77 standard constants using the apostrophe character (`'`) for quoting.

`-D`*name[=def]* Define the macro *name* to have the value *def*, as if MPPL had read the statement

```
define name def
```

This option may be repeated. Careful quoting is required to embed blanks into *def*:

```
-Dname="'this is a string'"
```

is typical.

-f Set free-form input. This disables the usual column 1 comment convention and the column 6 continuation convention. MPPL # comments may still be used in any column. If you only want to disable the column 6 continuation convention, specify a `-ccC*` (or similar `-c` option) after the `-f` option.

-i *N* Set the `IntegerSize` to *N*. Legal values are 2, 4, or 8, and imply that an *integer* variable without *kind-selector*, or literal integer constant will be stored in at least 2, 4, or 8 bytes, respectively.

-I *directory* Insert *directory* into the search path for include files. Usage is similar to the UNIX C preprocessor. For instance, the options

```
-I.. -I/usr/local/vbasis/pkg
```

tells MPPL to search the parent directory and `/usr/local/vbasis/pkg` for include files, in addition to the current directory. The current directory is always searched first.

-l (ell, for "long") Set the length limit for output lines to 80 instead of the standard 72 columns. This limit does not apply to comment lines.

-m Prevent MPPL from activating the Basis definitions. Non-Basis users of MPPL should use this option if problems develop from the Basis definitions.

-M*machine* Specify the machine we intend to compile on. This sets the value of the macro `MACHINE` to *machine*, and may affect the definition of other predefined macros.

-r*N* Set the `RealSize` to *N*. Legal values are 4, 8, or 16, implying that a *real* variable with no *kind-selector* or literal floating point constant of the form `0.0e0` will be stored in an element of at least 4, 8, or 16 bytes, respectively.

-t *Sys* Set the intrinsic macro `SYSTEM` to *Sys*, and set the value of other intrinsic macros to the default values for the system named. This allows you to "cross-compile" source for a Fortran compiler on another system. This option sets the macros `MACHINE`, `COMPILER`, `TYPE`, `CHAR_PER_WORD`, `LOCS_PER_WORD`, and `WORDSIZE` to the defaults for the target system. Use the `-C`, `-D`, or `-M` options to over-ride these defaults as required.

-u Provide "case insensitivity" for macro names. Either all upper or all lower case (not a mixture) may be used to invoke a macro. This option is required if Fortran keywords in your source code are in upper case.

-v Turn on verbose output. Note each input file as it is processed.

-w Turn on extra warning messages. In particular, warn if *Size* requests cannot be satisfied in the given target compiler. E.g., if the compiler has no 16 byte wide floating point type, then a request for a *Size16* real object will be mapped into *Size8*, and, if the flag was given, a warning message will be printed to *stderr*.

--langf77 Convert mppl language macros into Fortran 77. (default)

--langf90 Convert mppl language macros into Fortran 90. The output will be free source form.

--isf90 The source is already f90 free form. This will expect f90 style continuations.

--nolang Do not convert mppl language macros.

--nonumeric do not convert numbers from f90 format (1.0_Size8), do not process `-r8` or `-r4` macros ($1.0e00$ will not be converted to $1.0d00$) and do not read mppl.std which define integer, real and other related macros.

--macro Expand macros. This is the default behavior.

--nomacro Do not expand macros.

--pretty Pretty print i.e. indent lines. Each level of indention uses the `continuation-indention` value. This is the default.

--nopretty use existing white space.

--relationalf77 convert conditions to use f77 relations operators (.eq., .ne., ...) This is the default behavior.

--relationalf90 convert conditions to use f90 relations operators (==, /=, ...)

--honour-new-lines --honor-new-lines -hnl preserves existing line breaks with `--pretty` option.

--continuation-indention*n* -ci*n* The width to indent blocks and continued lines. Defaults to 3.

--comment-indentation*n* -comi The column to start embedded comments (comments using the # character). This is only valid with `--langf90`

The `-m` option, which must occur before the name of the first input file, prevents MPPL from activating the Basis definitions. Non-Basis users of MPPL should use this option if they find any problems develop from this change. Chances are pretty good that this is not really necessary, since if one of your own definitions collides with the built-in one it will replace it. To see the Basis definitions, run MPPL interactively and enter Dumpdef. Each pair of lines printed are a keyword and its definition. The keywords are:

```
CHAR_PER_WORD COMPILER DEFAULT DONE DYNAM Dumpdef Dynamic ERR FALSE
Filedes Filename GENERATE LOCS_PER_WORD MACHINE Module NO NOTSET
Number_of_Database_Words OK Pi Point Prolog
Quote SITE SLEEPING STDERR STDIN STDOUT SYSTEM TRUE TYPE
UP Use VARNAME WORDSIZE YES
_integer _real _complex _logical _character Ch _Ch _double _Filenam
_Filedes  _Varname
SS_WIDTH SS_N SS_TC SS_PTR SS_NAML SS_NS SS_N1 SS_M1 SS_I1
```

84.3 Token Processing

The first internal operation that MPPL performs is the collection of data units, or "tokens". Tokens, or strings of characters, are collected one a time. Some are passed directly to MPPL's output or "translation", and some are checked to see if they require expansion.

84.3.1 Token Descriptions

Alphanumeric An alphanumeric token is any sequence of letters and digits that begins with a letter. The underscore character (_) is treated as a letter.

Digits Tokens can be any one or more digits, 0–9.

Real Numbers Tokens can be Digits followed by a decimal point and exponent.

White Space Tokens can be any sequence of blanks and/or tabs.

Quoted String Tokens can be made up of Hollerith constants or Fortran strings in either single (') or double quotes ("). The same type of quote mark can be used inside a quoted string if the marks are doubled. Or the opposite type of quote may appear.

Comment Everything between a pound sign (#) or an exclamation point (!) and the end of the physical line is a comment. MPPL changes the first character to a lowercase c and writes the token to the output IMMEDIATELY. A new token is then collected. This means that a special method must be used to include comment lines in macro definitions. Refer to the description of the Immediate macro in "Macro Processing" below.

Logical Operators `.eq.`.

Multiple Character Operators exponentiation (**) and concatentation (//).

Any Other Single Character For example, a decimal point is a token. Note, however, that MPPL ignores (and discards) the backslash (\) and collects another token. If the last non-whitespace token on a line is a backslash, MPPL continues the line. The backslash is useful

for separating units that must be interpreted separately, but which the user wants adjacent in the output.

"Newline" The invisible "return" character at the physical end of a line of input text is recognized as a token we call "newline". In two cases, however, MPPL discards newline so that two or more physical lines can become one logical line: the column-6 continuation (the Fortran continuation convention) and assumed continuation.

In the Fortran continuation, the newline token and the first six characters of the next line are discarded if the next physical line begins with five blanks followed by a non-blank character.

Assumed continuation occurs when the last non-whitespace token on a line is `+, -, *, (, comma, &, |, ~, >, < , = or \`. The user may conveniently continue a long quoted string by adding a backslash to a concatenate operator (`//`), for example:

```
x = "This is a long string"//\
"divided into two parts"
```

Note that MPPL does not treat the forward slash (division) character as an obvious continuation because the forward slash is the final character in DATA statements.

84.3.2 Processing Traditional Comments

MPPL recognizes `c, C, or *` in column 1 of a physical input line as a standard comment line, and writes the entire line immediately to the compiler-ready output. The list of characters that signal comment lines may be altered by means of the `-c` ("minus c") option described above in "Specifying Options."

84.3.3 Free-Form Input

MPPL ignores positioning of statements on a line except for the column-6 continuation convention and the `c, C, or *` in column 1, the comment-line indicator.

84.4 Macro Processing

84.4.1 Basic Features of the Macro Processor

The second internal operation that MPPL performs is to replace the alphanumeric tokens the user has defined as macros with the appropriate text. The macro preprocessor collects any macro arguments, performs macro expansion and translation, generates labels, and then passes translated text to the statement processor.

MPPL macros have the following features:

- Recursivity (a macro can call itself).
- Easy-to-read, functional syntax resembles Fortran.
- Built-in conditional statement.

The built-in macros MPPL has are:

```
define(name,translation)
define name translation
Undefine([name])
ifdef([a],b,c)
ifelse(a,b,c,d)
Errprint(message)
Infoprint(message)
Dumpdef([macroname])
Immediate(argument)
Evaluate(argument)
Remark(message)
Setsuppress(name,char)
include filename
Module
Prolog
SYSTEM
```

The MPPL define macro lets users define their own macros. Macros have many uses; they can:

- Give symbolic names to constants, so global changes need be made in only one place.
- Conditionally compile blocks of code.
- Abbreviate or customize the language of frequently used blocks of coding where a subroutine call is not desired.
- Improve readability of the code to make its structure and purpose more obvious.

84.4.2 Macro Names

A macro name can be a string of alphanumeric characters (upper case and lower case letters, digits, and the underscore character) of any length. Note that the macro processor is sensitive to case. N and n are recognized as different names. The `-u` command line option can override this behavior.

84.4.3 Argument Collection

If a macro has arguments, the macro name is followed by a left parenthesis. Arguments are separated by commas and the argument or argument list is terminated with a right parenthesis. Commas within the second or deeper levels of parentheses, or inside square brackets, are ignored. Each argument in turn is collected, and each alphanumeric token is scanned to see if it is a macro. In the following example, the define macro has just two arguments, `"Jack"` and `"Jill(went,up,hill)"`:

```
define(Jack,Jill(went,up,hill))
```

If a macro name has been specified in a `Setsuppress` macro, then argument collection is supressed.

84.4.4 Macro Expansion

Because macro names are alphanumeric tokens (as defined above), every alphanumeric token must be checked. If a token is a macro name, its arguments (if any) are collected, and the expansion of the macro is "pushed back" onto the input file to be rescanned for tokens as described earlier in "Token Processing."

Square brackets are most often used around the arguments to macros. Macro expansion can be delayed by placing the macro name in one or more pairs of square brackets. Each time brackets are encountered, the outside pair is stripped off. For example:

```
define N 100
[N] = N
```

translates to

```
N = 100
```

In a second example, in line 2 below, the N is expanded to 12 when arguments are collected, so the first argument does equal the second. In line 3, the first argument is N, and the second argument is 12.

```
define(N,12)
ifelse(N,12,true,false)          = true
ifelse([N],12,true,false)          = false
```

84.4.5 Macro Translation

When a macro is invoked in the code, it is translated using information from the macro definition. The following substitutions are made:

- Argument substitution (`$n`).
- Replacement of `$*`.
- Replacement of `$-`
- Label generation (`@n`).

Argument Substitution

Any dollar sign followed by a digit 1–9 in the argument list in the define statement is replaced by the corresponding macro argument: `$1` is the first argument, `$2` the second, etc. `$0` is the name of the macro being expanded.

A dollar sign followed by an asterisk or a minus sign, is treated as explained below. A dollar sign followed by another dollar sign results in the insertion of a single dollar sign into the expansion text. A dollar sign followed by any other character results in the insertion of that other character into the expansion text.

```
define distance sqrt(($1-$3)**2 + ($2-$4)**2)
w = distance(x1,y1,x2,y2)
```

expands to

```
w = sqrt((x1-x2)**2 + (y1-y2)**2)
```

Replacement of $*

The complete argument list, separated by commas, is generated. Thus, if we define Jill as

```
define Jill hill($*) - $1
```

then the macro statement in the code

```
Jill(up,down)
```

is translated as

```
hill(up,down) - up
```

Replacement of $-

The argument list minus the first argument is generated. This can be used to define macros with an arbitrary number of arguments that process the first argument and then call themselves recursively to process the remaining arguments. For example:

```
define Product $1 REST($-)
define REST ifelse($1,,,[* $1 REST($-) ])
w = Product(x,y,z)
q = Product(x)
```

which expands to

```
w = x * y * z
q = x
```

The `ifelse` macro is explained below; the result is simply to terminate the recursion when there are no more arguments left. This is a hard example, but we present it because of the usefulness of the idea.

Label Generation

The combination of an at sign (@) followed by a digit 1–9 is replaced by an automatically generated label number. Each occurrence of @n is replaced by the same number within a particular expansion of the macro. The first number assigned is the next number in the automatic label sequence, as described in “Execution: Selecting Options.”

In the following example, square brackets protect the second argument of the define macro from token interpretation as it is collected. The expansion of the macro named `Errorif0` is given below. It is good practice to use the brackets. They usually produce the desired results, but in this case, they are not really necessary.

```
    define(Errorif0,[
    if ($1.ne.0) go to @1
    write(6,@2)
@2  format("$1 is zero.")
    return
@1  continue
    ])
    Errorif0(x)
```

expands to :

```
        if (x.ne.0) go to 23000
        write(6,23001)
23001           format("x is zero")
        return
23000           continue
```

When a macro is expanded, quoted strings do not protect any arguments (e.g., `$1, $2`) inside them. But when a quoted string is seen by the token processor, macro names inside will not be recognized by the macro processor.

84.4.6 User-Defined Macros

Users define a macro with the built-in MPPL macro define. The two forms of the define macro are:

```
define macroname expansion
define(name,expansion)
```

In the first form, the next token after the define is taken as the macro name. After skipping over any space following the name, MPPL takes the rest of the line as the expansion. Neither the name nor the expansion is scanned for further macros to expand.

In the second form, a define macro with arguments looks like a Fortran function call. The arguments are in parenthesis separated by commas. This form is treated like a normal macro invocation; the arguments are scanned as they are read. If name is already defined then to redefine it using the second form one must surround name with square brackets so that it is not expanded as it is read.

If name has already been defined, the old definition is forgotten. A macro name can be forgotten altogether with the `Undefine` macro.

Undefine([name])

The Undefine macro deletes the definition of a macro name. Note the required square brackets to prevent the name from expanding before we get a chance to Undefine it!

84.4.7 Built-in Macros

In addition to the define macro, the other predefined macros in MPPL are ifdef, ifelse, Errprint, Dumpdef, Immediate, include, COMPILER, SYSTEM, MACHINE, SITE, TYPE, Prolog, Errprint, Infoprint, and Module. The functions they perform cannot be accomplished with user-defined macros.

In addition, the higher-level constructs in MPPL are actually implemented as built-in macros. For example, there is a macro whose translation is a special nonprintable character that is interpreted at the statement level.

The built-in macros are described below.

ifdef Macro

```
ifdef([a],b,c)
```

is replaced by either `b` or `c`, depending on whether `a` was defined or not. It becomes `b` if `a` is a defined macro name, and expands to `c` if `a` was not a defined macro name (provided `c` is given). The name `a` needs to be protected with square brackets. For example,

```
ifdef([DEBUG],call trace("x",x))
```

ifelse Macro

If the first argument is identical to the second, the `ifelse` macro,

```
ifelse(a,b,c,d)
```

is replaced by the third argument. Otherwise, it expands to the fourth argument. The second argument `b` can be of the form `b1|b2` in which case, the equality is satisfied if `a` is identical to `b1` or `b2`. Using the notation above, `ifelse(a,b,c,d)` is read as `"if a = b, then c else d."` In making the comparison, leading and trailing spaces in `a` and `b` are ignored. An example of this macro is

```
define Dim real $1[]ifelse($2,,,($2))
Dim(x)
Dim(y,100)
```

which expands to

```
real x
real y(100)
```

The pair of square brackets in the definition of Dim is used as a token separator, so that `ifelse` will be recognized after the name is substituted for `$1`.

Errprint Macro

The Errprint macro immediately writes the argument to the user's terminal in the form MPPL:message and a bell rings. This message goes to the terminal, not to the output file. The syntax is

```
Errprint(message)
```

Infoprint Macro

The Infoprint macro immediately writes the argument to the user's terminal in the form MPPL:message and a bell rings. This message goes to the terminal, not to the output file. The syntax is

```
Infoprint(message)
```

Dumpdef([macroname])

If Dumpdef has no arguments, all macro definitions are displayed to the terminal. If there are arguments, the definition of each macro name given is written to the terminal. The macro name needs to be protected from expansion during argument collection by square brackets, as shown.

Immediate(argument)

Because the token processor writes comments out immediately, the Immediate macro is the best way to delay writing a comment line until it is wanted. For example,

```
define A_comment Immediate([c  this is a comment])
```

is written out as

```
c  this is a comment
```

when the translation for `A_comment` is rescanned. The argument of the Immediate macro is immediately written directly to the output file as a separate line without further interpretation. Note the square brackets surrounding the text of the above `Immediate`. They are recommended in order to suppress the expansion of any macro name, or MPPL reserved word, that might inadvertently been included in the comment.

Comments beginning with "*", "#", and "!" are discarded from macro text upon expansion.

Remark(argument)

The remark macro is used to insert a comment into the code. A limitation of using `Immediate` to insert comment occurs when switching from generating f77 fixed-form to generating f90 free-form. `Remark` will use the correct comment convention based on the `--langf77` and `langf90` command line options.

For example,

```
define A_comment Remark([ this is a comment])
```

is written out as

```
  c  this is a comment
```

when using the `--langf77` option; and,

```
  !  this is a comment
```

when using the `--langf90` option.

Evaluate(argument)

Evaluate calculates the value of the integer expression represented by argument and returns the character form of the result. If argument is not an integer expression then Evaluate returns argument itself. Example:

```
define N 22
define(NP1, Evaluate(N+1))
define(NP1S, Evaluate(N + 1.0))
x = NP1
y = NP1S
```

expands to

```
x = 23
y = 22 + 1.0
```

Note that in the expression for `y`, `Evaluate(N+1.0)` resulted in a call to Evaluate with argument `"22 + 1.0"` (since the argument was scanned for macros as it was collected), and since this was not an integer expression, Evaluate returned it verbatim.

include filename

The include macro inserts the contents of filename into the input stream. The statement causes the named file to be read before continuing to read the current input file. The included file may itself contain other include statements, to a depth of five files.

Setsuppress(name,char)

Setsuppress is used to suppress argument collection for a macro when it is followed by a specific character.

```
define RealSize  Size4
define(real,\
[ifelse(RealSize,Size4,[[real]([$*])],[[dble]([$*])])]\
)

real(b)
real*8 foo
Setsuppress([real],[*])
real*8 foo
```

expands to

```
      real(b)
      real()*8 foo

      real*8 foo
```

The `Setsuppress` macro prevents the `real` macro from being expanded when used in the `real*8` context.

CHAR_PER_WORD

CHAR_PER_WORD evaluates to the number of characters per machine word. Present machines have either 4 or 8 characters per word.

COMPILER

COMPILER evaluates to the name of the Fortran compiler we are planning to use.

LOCS_PER_WORD

LOCS_PER_WORD evaluates to the number of locations per machine word. Present machines have either 4 or 1 locations per word.

MACHINE

MACHINE evaluates to the name of the machine we are planning to use.

Module

Module evaluates to the name of the current subroutine, function or program module. It evaluates to ? if between modules or if in a main program which does not contain a program statement.

Prolog

After each subroutine, function, or program statement, MPPL adds a line containing the statement Prolog. Prolog is predefined to be simply a comment. The user may redefine Prolog in order to include certain statements in every subroutine and function, such as:

```
define Prolog implicit integer(a-z)
```

SYSTEM

SYSTEM evaluates to the name of the operating system on which MPPL is being run. Currently available systems include AXP,LINUX,LINUXA,HP700,SGI,IRIX64,SOL

WORDSIZE

WORDSIZE evaluates to the length of a word in bits. Currently available wordsizes are 32 and 64.

84.4.8 Error Messages

MPPL error messages are written both to the terminal and to the output file. Where possible, MPPL tries to continue processing after an error (e.g., an `endif` statement with no matching `if` statement). MPPL tries to begin again at the next physical line. As is common in such cases, one error may cause several error messages because the first error confuses MPPL.

Errors in the macro processor are often extremely difficult to handle, and many of these errors cause MPPL to halt immediately. Since the higher-level constructs are macros, mistakes involving their keywords can lead to errors that are reported as errors in the macro processor. For example, a missing right parenthesis in a `return` statement

```
return(value
```

leads eventually to an error as MPPL proceeds to eat up text looking for the end to the argument list for return. MPPL tries to help in this case by informing you that it was collecting arguments when the error occurred, and naming the macros involved.

84.5 Statement Processing

In statement processing, the third internal process, MPPL collects Fortran statements and writes them to MPPL's output file in standard form. During this operation, MPPL indents do loops and if-then statements, and continues long lines using the column-6 convention.

Another major part of statement processing is the transformation of the nonstandard constructions listed below into standard Fortran:

Looping Constructs

```
do ; ... ; enddo
do ; ... ; until (condition)
while ; ... ; endwhile
for( initial, condition, reinitial); ... ;  endfor
break (or break n)
next          (or next n)
```

Module Declarations and Function Value Return

```
subroutine, program  or function
return
return(value)
end
```

Conditional and Case Statements

```
if(condition) then;...; else ; ... ; endif
if(condition) return(value)
select(expression) case default endselect
symbols for logical operators: >, <, >=, <=, <> or ~=, = or ==
```

Free-Form Input

```
; is a logical newline
# and ! begin comments
Automatic continuation if line ends in +, -, *,  comma,
  (, &, |, ~ ,=,>,<
```

These extensions to the Fortran language allow the user to write programs with clearer structure and meaning, and to reduce the use of goto statements and labels.

The keywords listed above are macro names that are translated to special nonprintable characters recognized by the statement processor. When using these macro names, it is important to be aware of the considerations discussed below.

84.5.1 Cautions on the Use of Keywords

No Spaces in Macro Names

Do not include spaces within the names. Like all macro names, they cannot be separated internally. The statement

```
d o 100 i = 1,n
```

is not recognized as a do statement in MPPL, even though standard fixed-form Fortran allows the space. The user may separate `end do`, `end while`, `end for`, `end if`, and `end select`, however.

Error If Name Out of Context

These macro names cannot be used in other contexts (e.g., a variable named do is incorrect). If misplaced in the input, these macro names cause an error message, usually “Unprintable character or misplaced keyword in output.”

An expression in parentheses that follows one of these macro keywords is macroexpanded during argument collection, and is rescanned in cases where the argument is supplied. For instance, the built-in definition of `if` is not just a special nonprintable character `X`, but rather is `X($1)`. Understanding the way the keywords are seen internally is important, as the next example shows. Given

```
define n x
define x 10
```

then

```
 if ([n]>9) goto 70
```

translates to

```
 if(10.gt.9) goto 70
```

but

```
if([[n]]>9) goto 70
```

translates to

```
if(n.gt.9) goto 70
```

Protected Token Interpretation

The user should protect keywords with square brackets inside macro definitions to prevent early interpretation. For example,

```
define(zero_out,do i=1,n;$1(i)=0.;enddo)
zero_out(x)
zero_out(y)
```

will result in two do loops with the same label. Instead, to obtain the correct result, write the definition as

```
define(zero_out,[do i=1,n;$1(i)=0;enddo])
```

84.5.2 Symbols for Logical Operators

In the `if`, `for`, `while`, and `until` statements you can use symbols for the standard logical operators (e.g., `<` for .lt., `>` for `.gt.`). The complete list of acceptable symbol substitutions is given below in “Conditional Statements.”

84.5.3 Multiple Statements on a Line

MPPL treats a semicolon (`;`) as a logical newline only. Note that column-1 conventions only refer to physical lines. Thus, in this example, a `c` that follows a semicolon is not the start of a comment. Also, as shown here, a label is allowed in the middle of a line:

```
x=0;c=0;100 format(i5)
```

Of course, just because you can do something doesn’t mean you should.

84.6 Looping Constructs

84.6.1 Do Loops

do i=1,n;...;enddo

The `do-enddo` construct is available in addition to the traditional `do` loop of the form

```
      do 100 i = 1,n
100   continue
```

The user omits do-loop labels (100 in the example above) and MPPL supplies them during creation of compiler-ready output. The user may specify the lowest number with which MPPL begins numbering (the default is 23000; see “Execution Options”). The syntax is

```
do i=1,n
. . .
enddo
```

The numbering sequence restarts at the beginning of each module.

do/enddo

MPPL allows a `do/enddo` loop without a variable, which is a "do forever" construct with the form

```
do
. . .
enddo
```

In this construct, MPPL generates a labeled continue statement on the `do` line, and replaces `enddo` with a `go to` statement transferring back to that label. The user must provide an exit within this loop by means of a `go to` statement, a `return` statement, or a `break` statement. The last three statements are explained later in this section.

do/until

The user may also select the `do/until` construct, which causes the loop to repeat until the condition given is true:

```
do
.
.
.
until(condition)
```

Note that the body of this loop is always executed at least once.

84.6.2 While Loops

A `while` loop allows the user to repeatedly execute a block of statements while the condition remains true (e.g., while an error is too large, or a desired element has not been found in a table). This statement replaces the traditional `do` loop with an `if`-test/`goto` inside it. The condition is tested at the top of the loop:

```
while(condition)
 . . .
endwhile
```

For Loops

The `for` loop (modified from the for loop in the C language) is a versatile construct that handles many problems not suited to processing by ordinary looping constructs. It is useful for loops in which the changing element is not merely incremented, but rather may be a call to a function, multiple statements, or another nonlinear process. Note the use of commas instead of the semicolons used in C. The second argument, the condition, must always be given. The third argument is optional.

```
for(initial,condition,reinitial)
. . .
endfor
```

MPPL translates the construct as shown below. First, the initial clause is executed, and then the condition evaluated. If the condition is true, the body of the loop is executed. Then the reinitial clause is executed, and the condition reevaluated. The loop terminates when the condition becomes false.

```
   initial
   go to L3
L2 reinitial
L3 if(.not.(condition))go to L1
   . . .
   go to L2
L1 continue
```

The first and third arguments may contain multiple statements, and the first argument can be null, for example,

```
for(,n<10,n=n+1)
  i = i + n
endfor
```

84.6.3 Leaving and Skipping

break

The MPPL `break` statement can be used inside any of the looping constructs discussed above. It is invoked in any one of three forms:

```
break
break(n)
break n
```

where `n` is an integer that specifies the number of loops from which a breakout is desired. The `break` statement translates to a `go to L` statement, where `L` is the supplied label of a continue statement that follows the end of the loop. If $n > 1$, the transfer is to the end of the n'th enclosing loop, e.g.,

```
do i = 1,10
   do j = 1,10
        if(x(   i,j).eq.0)break 2
   enddo
enddo
```

Here, the `break 2` statement causes a transfer out of both loops. If the 2 is omitted, transfer would be just out of the j loop.

next

The `next` statement can also be used inside any of the looping constructs. It causes the next iteration of the loop to begin.

The slight differences in implementation for each kind of loop are shown below:

Type of Loop	Go to:
Traditional do loop	labeled statement
Label-less do loop	enddo
do/enddo, do/until	do
while/endwhile	while
for/endfor	reinitial

Note that, in each case, the transfer is to the top of the loop. However, the labeled loops go to the label to increment the variable. In traditional loops, where the labeled statement is not continue, the labeled statement is executed, which may be surprising. Note also that the `do/until` loop executes the loop body at least once after the use of a `next` statement.

84.6.4 Module/Return Statements

Modules may begin with standard `program`, `subroutine`, or `function` statements. These three words are MPPL macro names so that MPPL can issue good error messages and so that functions can return a value from a function in a more natural way.

Inside a function, the user may provide an argument to the `return` statement:

```
return(value)
```

MPPL expands this to

```
functionname = (value)
return
```

It is an error to use an argument with the `return` statement inside a program module or subroutine module. In that case, MPPL displays an error message, but continues execution. A statement of the following form is allowed:

```
if(condition) return(value)
```

84.6.5 Conditional Statements

MPPL supports all the standard `if` and `if-then-elseif-else-endif` constructs of Fortran 77. It also adds some extra features to these statements.

Symbol Substitution

In addition to processing Fortran 77 forms of the `if` statement, MPPL allows the user to enter the following symbols for equals, greater than, etc., instead of the traditional notation.

User enters	translation
`>`	`.gt.`
`>=`	`.ge.`
`<`	`.lt.`
`<=`	`.le.`
`~=`	`.ne.`
`<>`	`.ne.`
`~`	`.not.`
`=`	`.eq.`
`==`	`.eq.`

if(condition) enhancement

MPPL also allows placing the last part of an `if`(condition) statement on a new line. For example:

```
if ( ierr > 0 )
   call goof
```

or

```
if ( ierr > 0 )
then
   call goof
endif
```

if(condition) return(value) statement

This special single-statement

```
if (condition) return(value)
```

appears to be a statement of the form

```
if (condition) statement
```

However, `return`(value) translates to two statements. MPPL handles this in a special way in order to translate it correctly to:

```
if(condition) then
  functionname=(value)
  return
endif
```

84.6.6 Case Selection Statement

The syntax for the select macro is:

```
select(expression)
case casenum:
. . .
default:
. . .
endselect
```

where

```
select(expression)
```

compares an integer expression to the values listed in the case statements that follow, and executes at most one of the cases. The first `case` stated must immediately follow the select statement. An optional `default` section can be executed if the expression fails to match any of the `case`s.

```
case    casenum:
```

labels the beginning of the statements to execute if the select expression matches the case expression casenum. For casenum, the user must insert either an integer, a range (two integers separated by a minus sign), or a comma-delimited list of integers and ranges. The expression must end with a colon. For example:

```
case 7-10,12:
```

Statements may follow on the same line, after the colon. Multiple statements may be separated by a colon, or appear on new lines.

```
default:
```

labels the beginning of the statements to execute if the select expression fails to match any of the case values.

```
endselect
```

marks the end of the case list. Here is an example of a complete select/case/default/endselect construction:

```
select(x)
case 0: y = 1
   x = 1
case 1-4: y = 2
case 5,6: y = 3
case 7-10,12:
   y = 4
default: y = 0;x = 0
endselect
```

A `select` statement is translated either into a series of `if` statements or into a computed go to. The latter is more efficient and so is used if there are enough consecutive case values to make it desirable. A few gaps in the sequence will be filled in and the sequence need not start from one. A computed go to is a statement of the type

```
go to (1000,1001,1002, ...) ivar
```

where control goes to label 1000 if ivar = 1, to label 1001 if ivar = 2, etc. While efficient, such statements are opaque, annoying to modify, and have undefined behavior if ivar is out of bounds. The select statement is both clearer and safer.

84.7 Sample Input File Showing Major MPPL Features

```
#LOOPING CONSTRUCTS
#
define N 100
define M  20
   function shoot(j)
c This subroutine shows the six different looping constructs
   real xx(N),y(M),x,y
# there are four kinds of DO loops plus WHILE and FOR loops.
#

# TRADITIONAL LABELED DO LOOP
   do 100 i=1,10
        if(x(i) = 4) then
             break         # same as go to next stmt after 100
        endif
        if(x(i) = 5 then
             next # same as go to loop label (100)
        endif
100          y(2) = x(i)         # this gets executed on a next
#

# DO LOOPS WITHOUT LABELS
# next gets you to next iteration; break gets you out
# SIMPLE LOOP
#
   do i=1,M
        if(y(i) < 0) break
        if(y(i) >= 10.) next
        y(i) = sqrt(10.-y(i))
   enddo
#

# NESTED LOOPS
#
   do i = 1,M
        do j = 1,N
             if(x.eq.10)then #next iteration of inner loop
                  next
             endif
             if(x.eq.20)then #next iteration of outer loop
```

```
                    next(2)
               endif
               xx(j) = 8
               if(y(i) > x(j))then
                    break     # get out of inner loop
               else
                    break(2)  # get out of inner loop
               endif
          enddo j   # end inner loop
                    # ignores anything after enddo
     enddo i            # end outer loop
#


# DO FOREVER
# repeats forever; get out with break, return, or goto.
#
     i=0
     do
          i = i + 1
          if(i > M) break
          if(x(i) == 32)
               next
          x(i)=1/(x(i)-32)
     enddo
#


# WHILE/ENDWHILE
# does a loop as long as the condition is satisfied
#
     i = N
     while(x(i)-x(i+1) > 1.e-5 & i <> 0)   #&=.and.  <>=.ne.
          i = i - 1
     endwhile
#


# DO/UNTIL
# repeats until the condition is satisfied. Note that unlike
# a while loop, the loop body is always done once
#
     do
          i = i - 1
          if(i = 0 | x(i) <= 0.) break        # | = .or.
     until(x(i)-x(i+1) < 1.e-3)
#
```

```
# FOR/ENDFOR
# has three arguments separated by commas:
# a) initialization statements to be executed before the
# loop, b) the condition under which the loop is to be
# executed while true, and c) the reinitialization
# statements to be executed at the start of each loop after
# the first before the condition is tested. The condition,
# argument 2, must be present; other arguments are optional.
#
# The following example is the same as do i=1,N;x(i)=i;enddo
#
   for (i-1, i<=N, i=i+1)
        x(i) = 1
   endfor
#
# FOR loops are good for things DO LOOPS can't do:
# the hard way to find the square root of two is:
#
   for(t = 1.,abs(t**2 - 2.) > 1.e-6, t=(t+2./t)/2.)
   endfor
#

# FUNCTIONS
# The return statement can have an argument to give the
# returned value.
#
   return(t)
   end
   real function boxo(w,z)
   real w,z,a,b
#

# IF STATEMENTS
# There are two basic kinds of IF; this routine shows some
# of the variations allowed.
#
# IF(CONDITION) THEN ...ENDIF
#
   if( a<b)            #ok if then is on next line
   then
        call odd("this is a string; try it");return(b-a)
   endif
   if(a <> b) then         #if a .ne. b
        x = y
   else if (b > a-1)         #ok if you forget the then here
```

```
        x=y/2 +         #statements continued if they end
        golf(tango,         #in +,-,*,comma,=,(,&,|,caret, or
        bravo \         #backslash; backslash is deleted
        -1)
        y="This is a quoted string "" with a quote in it\"
                       #...but not inside strings
   else if("the sky is blue >")then #or put in to be neat
        howdy = 1
   else
        if(a == w) call junko
   endif
#


# IF(CONDITION)STATEMENT/RETURN(VALUE)
# is correct even if it expands to more than one statement.
#
   if(a > b)
        b = b/2
   if( a<> b) return(gas)
   return(bug)
   end
   program testme
#


# SELECT/CASE/DEFAULT/ENDSELECT
# You can put things after an ENDDO that are ignored.
#
   real x(N)
   do i=1,10
        x(i) = i - 1
   enddo i --end of loop setting initial values for x
#
# You can have multiple statements by separating them
# with semicolons, even in the arguments of a FOR statement.
#
   i0 = 0 ; j0 = M
   for( i = i0; j = j0 , j < 9 , i=i+1;j=j-1)
        x(i) = y(j)
        for(k=j, k<i+5 , k=k+1)
             z(k) = y(j) + x(i)
        endfor
   endfor
#
```

```
# SELECT allows you to test an integer variable against
# different cases.
#
   select(j)
   case 5: y=5         #if j is 5 do these two statements
        z = 4                ! exclamation points are also comments
   case 6: y=6         ! if j is 6 do this one
   case 7,8,10:         ! statements can follow on next line
        y=8;z=4         ! if j is 7, 8, or 10
   case 11-20,9: y=9         #if j is between 11 and 20
                       #inclusive or is 9
   default:
        y=0                 #do if j is none of the above
   endselect
#
   call exit
   end
```

84.8 Examples of Advanced MPPL Macro Usage

The following examples show how to use the macro processor. Most MPPL users will use macros only in the simple sense of using a name as a symbol for a constant value, as in

```
define pi 3.14159
```

and as in the first example below, to enable the specification of variables to be confined to just one place. Another common problem is conditional compilation, which we cover in the second example. The third and fourth examples show a user inventing language extensions.

84.8.1 Specifying a common block

This example shows how to specify a common block in one place, then use it as needed in subroutines. We include an Immediate comment so that in the expanded source the common block is marked with a comment.

```
define(Distribution_parameters,[
Immediate([c Distribution variables])
     integer alpha,sigma,beta
     common /c1/ alpha,sigma,beta
])
```

```
     subroutine x
Distribution_parameters
     . . .
     end
     subroutine y
Distribution_parameters
     . . .
     end
```

84.8.2 Conditional compilation

Depending on whether or not the first define(DEBUG,) line is present or not, the write statement is or is not compiled.

```
define(DEBUG,)
ifdef([DEBUG],[
     write(6,100) x,y,z
])
```

84.8.3 Vector operations

The following example shows a macro that expands to a do-loop that adds the last two arrays together and stores the result in the first array. The fourth argument is the length of the arrays. I do not advocate this kind of programming but it can be done.

```
define (Vector_add,[do i=1,$4 ; $1(i)=$2(i)+$3(i) ; enddo])
Vector_add(a,b,c,n)
Vector_add(d,e,f,n)
```

84.8.4 Alphanumeric Labels

Some people enjoy the LRLTRAN feature of using names as labels. This can be done with MPPL as long as we use a macro to change the names into statement labels. The Label macro is recursive so that several labels can be specified at once. The definition for Label can be read: if Label is called with an empty argument list, do nothing. Otherwise, define the first argument (`$1`) to be a macro name standing for the next available label (`@1`) and then apply Label to the rest of the arguments (`$`). Thus Label chews its arguments from left to right. Note that the `$1` is surrounded by square brackets in case this name was used as a label already in another subroutine.

```
define Label ifelse($1,,,[define([$1],@1)Label($-)])
```

```
      function boom(x)
c return 1, 0, -1 depending on the sign of x
      integer boom
      real x
Label(Negative,Positive) #must appear before first use of names
      if( x < 0) go to Negative
      if( x > 0) go to Positive
      return(0)
Positive  return(1)
Negative  return(-1)
      end
```

Conversion to MPPL

Those users who want to convert a code to precompile with MPPL instead of Precomp, but who do not plan to utilize the rest of Basis will have to make simple changes in their cliches. If a cliche is called `Abc` change the statement `cliche Abc` to `define{[UseAbc],[` and change the statement `endcliche` to `])\`. In the source every `use Abc` must be replaced by `Use(Abc)`. Basis does not support `dif` and `.if` directives. Replace them with combinations of the Basis macros `ifelse` and `define`.

84.9 Migration to Fortran 90 syntax

In the years since MPPL was first written, the Fortran standard has advanced to where the language processing features of MPPL can be replace by Fortran 90 syntax.

84.9.1 Command Line Options

A typical mppl-compile-load sequence is:

```
mppl mymacros mysource.m > myout.f
f77  myout.f -o xec
```

Often, the input file mysource.m and the output file myout.f are significantly different. All macros and real numbers have been processed and the output has been indented to a consistent form.

A line similar to

```
mppl --langf90 --nomacro --nonumeric --nopretty -l78
    mysource.m > mysource1.m
```

can be used to convert only the language macros.

The `--nolang` command line option can then be used to prevent the future expansion of MPPL language constructs.

```
mppl --nolang mymacros mysource1.m > myout.f90
f90  myout.f90 -o xec
```

84.9.2 Statement Processing

The `--langf90` option will produce free-form output. All comments start with an exclamation point (!). Embeded comments will replace the # with `!` without creating a new line. Continued lines end with an ampersand (&).

By default, f77 compatiable relation operators are used. `--relational90` can be used to generate symbols $<$, $<=$, $==$, $/=$, $>$, and $>=$

84.9.3 Macros

`include filename` is process by mppl. *filename* is read by mppl and processed. `include "filename"` is process by f90. *filename* is ignored by mppl.

The `Remark` macro should be use instead of `Immediate` to insert comments from macros. This will use the correct comment convention.

84.9.4 Loop Constructs

Indexed Loops

```
do i=1,n
...
enddo
```

This loop requires no conversion since it is valid f90.

do/until

```
do
  ...
until(condition)
```

`do/until` requires an explicit `exit`.

```
do
  ...
  if (condition) exit
enddo
```

While Loops

```
while(condition)
 . . .
endwhile
```

The `endwhile` is replaced with `enddo`.

```
while(condition)
 . . .
enddo
```

For Loops

```
for(initial,condition,reinitial)
  . . .
endfor
```

The `initial`, `condition` and `reintial` clauses are moved to the appropriate parts of a `while` loop.

```
initial
do while (condition)
  . . .
  reinitial
endfor
```

84.9.5 Leaving and Skipping

`next` and `next` are replace by `cycle` and `exit`.

The `next 2`, syntax is converted to use `goto`'s as with `--langf77`. A motivated user can manually convert this to:

```
outer: do
         do
           ...
           exit outer
         enddo
       enddo outer
```

84.9.6 Case Selection Statement

```
select(expression)
case casenum:
. . .
default:
. . .
endselect
```

`casenum` is enclosed in parenthesis. `default` becomes `case default`.

```
select(expression)
case (casenum)
. . .
case default
. . .
endselect
```

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z