The Basis System, part 6

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.

CHAPTER

TWO

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

THREE

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

FOUR

CTL: Package Control

4.1 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.

4.2 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.

4.3 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 4.1: 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.

4.4 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.

4.5 Adding CTL to Your Program

This section contains instructions for authors on how to add the `ctl` package to their program.

4.5.1 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.

4.5.2 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)
```

4.5.3 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.

4.5.4 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).

4.5.5 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.

4.5.6 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

FIVE

FFT: Fast Fourier Transforms

5.1 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.

5.2 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.

5.2.1 *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

$$z_k = \sum_{j=0}^{N-1} x_j \exp\left(\frac{-2\pi ijk}{N}\right), \tag{5.1}$$

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

$$c_k = \sum_{j=0}^{N-1} \left[x_j \cos\left(\frac{2\pi jk}{N}\right)\right] \tag{5.2}$$

, and

$$s_k = -\sum_{j=0}^{N-1} \left[x_j \sin\left(\frac{2\pi jk}{N}\right)\right] \tag{5.3}$$

.

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*,

$$y_j = z_0 + (-1)^j z_{N-1} + \sum_{k=1}^{l-1} 2\left[z_{2k-1}\cos\left(\frac{2\pi jk}{N}\right) - z_{2k}\sin\left(\frac{2\pi jk}{N}\right)\right] \tag{5.4}$$

for *N* even, *j*=0,...,*N*-1, For *N* odd, the term with the factor (-1)*j* does not arise.

5.2.2 *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

SIX

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

SEVEN

The History Package h2

7.1 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.

7.2 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.

7.2.1 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.

7.3 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.

7.4 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.

7.5 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
```

7.6 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.

7.7 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”.

7.8 Examples

7.8.1 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.

7.8.2 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_`.

7.8.3 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
```

7.8.4 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)'
```

7.8.5 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.

7.8.6 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)
```

7.8.7 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)
```

7.8.8 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

EIGHT

PFB Package

8.1 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 8.8.) 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*

8.2 Reading Files

8.2.1 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.

8.2.2 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.

8.2.3 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`.

8.2.4 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).

8.2.5 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.

8.3 Writing Files

8.3.1 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.

8.3.2 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 @.

8.4 Restoring From A FIle

8.4.1 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
```

8.4.2 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.

8.4.3 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.

8.5 Time Histories

8.5.1 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"`.

8.5.2 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*.

8.5.3 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.

8.6 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`.

8.7 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)

8.8 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.

8.9 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.

8.9.1 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.
```

8.9.2 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
```

8.9.3 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.
```

8.9.4 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
```

8.9.5 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
```

8.9.6 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

NINE

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

TEN

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

ELEVEN

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.

11.1 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)*.

11.1.1 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.)

11.1.2 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.

11.1.3 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.

11.1.4 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.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

W

Z