The Basis System, part 5

The Basis Development Team

November 13, 2007

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

UCRL-MA-118543

CONTENTS

1 The Basis System **1**
1.1 Environment Variables 1
1.2 Basis Is Both a Program and a Development System 1
1.3 About This Manual 2

2 Basis Development Overview **5**

3 Installing Basis **7**
3.1 Install Overview 7
3.2 Build Details 7

4 Dsys: Automating Building and Testing **9**
4.1 Dsys Targets 9

5 MIO: Make is OK **11**
5.1 Mio Overview 11
5.2 MIO output files 12
5.3 MIO syntax 14
5.4 Global Variables 17
5.5 System Group 18
5.6 Define Group 19
5.7 Setenv Group 19
5.8 Compiler Groups 19
5.9 CGroup Group 20
5.10 FGroup Group 20
5.11 LDGroup Group 21
5.12 LibGroup Group 21
5.13 Mac Group 21
5.14 Directory Group 22
5.15 File Group 24
5.16 Package Group 25
5.17 Archive Group 25

5.18 Library Group . . . 25
5.19 Program Group . . . 25
5.20 BasisProgram Group . . . 25
5.21 Fparse Group . . . 26

6 Getting Started Writing Packages **27**
6.1 Outline of the Process . . . 27

7 A Complete Example **31**
7.1 Overview . . . 31
7.2 Variable Description File . . . 31
7.3 config input File . . . 32
7.4 mio input Files . . . 32
7.5 Compiling and Loading . . . 33
7.6 Changing to Dynamic Memory . . . 34

8 Compiling Basis Packages **37**
8.1 Single Package Example . . . 37
8.2 Adding a Second Package . . . 41

9 Writing Basis Packages **45**
9.1 Basis Packages . . . 45

10 Precision and Portability **47**
10.1 Description of the Problem . . . 47
10.2 Specifying Precision in the Source . . . 47
10.3 Making Your Source Portable . . . 48

11 Fcc: Fortran Calls C **51**

12 Mac and the Variable Description File **53**
12.1 Sample Variable Description File . . . 53
12.2 Structure of the File . . . 54
12.3 Parameters . . . 54
12.4 Group Information . . . 56
12.5 Variable Descriptions . . . 58
12.6 Limiting Array Sizes . . . 59
12.7 Compileas Option . . . 60
12.8 Functions . . . 60
12.9 Making Arguments Optional . . . 61
12.10 Commenting the Variable Description File . . . 62
12.11 User Defined Types . . . 63
12.12 Architecture-dependent information . . . 64
12.13 Interfacing with C and C++; The Fcc Utility . . . 65
12.14 Writing Your Source . . . 67

13 Gluepack: Putting Packages Together **71**
13.1 config Execute Line 71
13.2 config Input File Format 71
13.3 Configuring the Packages with .pack files 75
13.4 config Errors 77

14 Programming Support Facilities **79**
14.1 Specifying Variables' Names 79
14.2 Dynamic Dimensioning 79
14.3 Output Routines 84
14.4 Replaceable Routines 89
14.5 Symbolic Constants 91
14.6 Symbolic Types 91
14.7 Physics Unit Codes 92
14.8 Interfacing with C and C++ Programs 93
14.9 Communication Between Packages 93
14.10 The Package Library 94

15 Advanced Package Writing **95**
15.1 There Be Dragons Here 95
15.2 Accessing Variables from Compiled Routines 95
15.3 Writing Attribute Services 97
15.4 Basis Supplied Servers 103
15.5 Writing Built-in Functions 104
15.6 Foreign Packages 111

Index **119**

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 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

THREE

Installing Basis

3.1 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
```

3.2 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

FOUR

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.

4.1 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

FIVE

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 methodl-ogy as simple as possible.

5.1 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.

5.2 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.

5.2.1 configured

A summary of the final configuration. Has all C¡Use¿ statemtents expaned.

5.2.2 configured.pl

A perl readable version of the config database.

5.2.3 code-m-def.d

Used by mppl source. creation controlled by `Write_m_defs`. Defines controlled by `VMDef`.

5.2.4 code-f-def.d

Used by fortran source. Creation controlled by `Write_f_defs`.

5.2.5 code-c-def.d

Used by C source. Creation controlled by `Write_c_defs`. Defines controlled by `VCDef`.

5.2.6 make-config

Used by **pck** to build the final makefile. Creation controlled by `Write_make_config`. Defines controlled by `VMake`.

5.2.7 Makefile

Global Makefile used to compile code in parallel. Creation controlled by `Write_makefile`.

5.2.8 mio.csh

Used by csh. Creation controlled by `Write_mio_csh`. Setenv controlled by `VEnv`.

5.2.9 mio.make

Used by make. Creation controlled by `Write_mio_make`. Setenv controlled by `VEnv`.

5.2.10 mio.pl

Used by perl. Creation controlled by `Write_mio_pl`. Setenv controlled by `VEnv`.

5.2.11 mio.sh

Used by bourn shell. Creation controlled by `Write_mio_sh`. Setenv controlled by `VEnv`.

5.2.12 packages

List of package names. Creation controlled by `Write_packages`.

5.2.13 Packages

Package groups for use by other codes. Creation controlled by `Write_Packages`.

5.2.14 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.

5.2.15 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.

5.3 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.

5.3.1 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.

5.3.2 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--`.

5.3.3 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.

5.4 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.

5.5 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

5.6 Define Group

Variables in this group are written out as macros.

5.7 Setenv Group

Variables in this group are written out as environmental variables.

VEnv Order to write out variables.

5.8 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.

5.9 CGroup Group

Variables for .c files. The global variable `default_CGroup` can be used to set the default CGroup to use to compile.

5.10 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*.

5.11 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

5.12 LibGroup Group

For building archives.

ARFlags

LibFlags

5.13 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.

5.14 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.

5.15 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`.

5.16 Package Group

System System group associated with this package.

5.17 Archive Group

5.18 Library Group

5.19 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.

5.20 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

5.21 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

SIX

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.

6.1 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

SEVEN

A Complete Example

7.1 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.

7.2 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
```

7.3 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> "
```

7.4 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.

7.5 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.

7.6 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

EIGHT

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

8.1 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.

8.1.1 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.

8.1.2 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> "
```

8.1.3 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.

8.1.4 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.

8.1.5 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.

8.2 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.

8.2.1 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
```

8.2.2 (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.

8.2.3 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.

8.2.4 Reconfigure

Run `mio` to reconfigure all the packages as well as the main code.

```
cd ~/foo/builder
mio -a lnx
```

8.2.5 Rebuild

Now to build the code do the following:

```
cd ../b
pck build
cd ../a
pck build
```

CHAPTER

NINE

Writing Basis Packages

9.1 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

TEN

Precision and Portability

10.1 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.

10.2 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.

10.3 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

ELEVEN

Fcc: Fortran Calls C

CHAPTER

TWELVE

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.

12.1 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
```

12.2 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.

12.3 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.

12.4 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.

12.4.1 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.

12.4.2 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.

12.5 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.

12.6 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 `+`, `-`, `*`, `/`.

12.7 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.

12.8 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".

12.9 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
```

12.10 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.

12.11 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.

12.12 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.

12.13 Interfacing with C and C++; The Fcc Utility

12.13.1 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" struct`s 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.

12.13.2 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.

12.13.3 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.)

12.14 Writing Your Source

12.14.1 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`).

12.14.2 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.

12.14.3 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.

12.14.4 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

THIRTEEN

Gluepack: Putting Packages Together

13.1 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 ˆ{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.

13.2 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.

13.2.1 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.
```

13.2.2 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.

13.2.3 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.

13.2.4 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
```

13.3 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.

13.3.1 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.

13.3.2 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.

13.4 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

FOURTEEN

Programming Support Facilities

14.1 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.

14.2 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.

14.2.1 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.

14.2.2 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`.

14.2.3 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.

14.2.4 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`.

14.3 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.

14.3.1 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`.

14.3.2 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.

14.3.3 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.)

14.3.4 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.

14.4 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.

14.4.1 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
```

14.4.2 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
```

14.4.3 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
```

14.4.4 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
```

14.4.5 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
```

14.4.6 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.

14.5 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.

14.6 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`.

14.7 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]
```

14.8 Interfacing with C and C++ Programs

See the chapter "Writing Basis Packages" for details.

14.9 Communication Between Packages

14.9.1 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?

14.9.2 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.

14.10 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

FIFTEEN

Advanced Package Writing

15.1 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.

15.2 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.

15.2.1 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
```

15.2.2 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 ze
        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.

15.2.3 Changing a package name to a number

```
function glbpknum(pn)
#find number of package whose name is pn
integer glbpknum
character*(*) pn
```

15.3 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`]).

15.3.1 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"`.

15.3.2 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.

15.3.3 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.

15.3.4 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.

15.4 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`.

15.5 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.

15.5.1 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 elem
      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
```

15.6 Foreign Packages

15.6.1 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`.

15.6.2 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.

15.6.3 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
```

INDEX

Symbols

. . . . 56, 58, 62
$ 56, 62
. . . . 54

A

actor 103
allot 81
ARCH 24
arg_coerce 106
arg_fetch_actual 105
arg_fetch_copy 105
arg_fetch_default 105
arg_fetch_fin 105
arg_fetch_init 105
arg_fix_dim 106
arg_get_address 105
arg_get_integer 106
arg_get_length 106
arg_get_name 106
arg_get_shape 106
arg_get_type 106
arg_kill 106
arguments
    optional 61
arrays
    dynamic 79
    dynamic$endrange> 84
    limiting 59
    partially full 59
    setlimit;setlimit 59
    temporary 83
attredit 58
attribute expression 98
attributes 57, 59, 97, 103
attrlist 58
autodyn 97

B

baderr 86
basclose 87
basfree 82
Basis
    data types 2
    documentation 2
    overview 1
    parser 2
basisech 89
basiserr 89
basiskit 27
basopen 87
baspecho 85
baspline 85
basterm 90
basurg 90
basusr1 90
basusr2 90
baswline 85
built-in 61
Burow, Burkhard 93

C

C and Fortran 93
C Language modules 93
C++ Language modules 93
cfortran.h 93
change 82

codefile 77, 87
codename 76
comments
  in variable description file 56, 62
  used to label output 62
compileas
  variable attribute 60
config
  array assignment 73
  array variables 76
  errors 77
  execute line 71
  foreign packages 114
  input format 71
  iotable 91
  package statement 72
  package statement example 73
  sample input 32, 114
  scalar assignment 73
  tokens 71
config¡$endrange> 78
continuation
  long function declarations 60
conversion
  name conflicts 91
  unit numbers 91
cprompt 76

D

data loading 62
differential compilation 24
dsys 9
  targets for dsys 9
    build 9
    commit 9
    config 9
    dist 9
    help 9
    info 9
    install 9
    link 9
    remove 9
    sync 10
    test 10
Dynamic 105
dynamic dimensioning 79
dynamic dimensioning$endrange> 84

E

echo 76
edit 88
environment variables 1
  BASIS_ROOT 1
  DISPLAY 1
  MANPATH 1
  NCARG_ROOT 1
equivalence statement 59
errors
  gluepack 77
external 60
EZN 2

F

Filedes 63, 91
Filename 63, 91
files
  closing 87
  opening 87
firstpkg 77
foreign packages 111
Fortran and C 93
Fortran intrinsics
  precision 48
freeus 87, 91
functions
  arguments
    optional 61
  as arguments to compiled functions 60
  built-in
    declaring 60, 61
    writing 104
  declaring compiled 60
  long calling sequences 62
  optional arguments 61

G

gchange 83
gfree 83
glbtmdat 90
gluepack 71

scalar variables . . . 76
short tutorial . . . 76
glurpack
sample input . . . 75
group
defining . . . 56

H

hexadecimal constants . . . 55

I

initialization
routine . . . 68
initialize
dynamic array space . . . 79, 81, 83
variables . . . 58, 62
iooutus . . . 85
iotable . . . 77, 91

L

limited
variable attribute . . . 59
list . . . 88
local
group attribute . . . 56
log
terminal . . . 84, 85

M

MIO . . . 11
mio . . . 11
adding a second package . . . 41
input . . . 11
simple Package file . . . 39
single package example . . . 37
MIO¡$endrange> . . . 24

N

names . . . 79

O

octal constants . . . 55
osallot . . . 83
oschange . . . 83
osfree . . . 83
output
Basis command . . . 85
terminal . . . 84, 85

P

package
foreign . . . 111, 117
naming . . . 45, 72
the .pack files . . . 75
padding . . . 81
parameter
defining in vdf . . . 54
expressions . . . 55
section in vdf . . . 54
parfind . . . 96
parget . . . 106
path . . . 77, 87
pathadd . . . 88
Point . . . 105
probname . . . 76
Prologue . . . 48

R

ranf . . . 47
Real4, Real8 . . . 47
remark . . . 85, 105
reserved words . . . 45
rtcntsiz . . . 103
rtcount . . . 103
rtfinder . . . 96
rtserv . . . 97
action string . . . 100
server string . . . 98
temporary variables . . . 101
rtxdb . . . 96

S

SC . . . 93
search path . . . 87
search stack . . . 79
servers . . . 58, 97
short name . . . 72
Size4, Size8 . . . 47
Smaug . . . 95
startup . . . 77

steerable applications 2
subroutines
    declaring compiled 60
sx_set_ndim 106
sx_set_shape 106
sx_set_type 106
symbolic
    constants 91
    types 91

T

terminal 84, 85
    log 85
tokens
    gluepack 71
types
    user defined 63

U

unit numbers 91
Use statement 29
user defined types 63
usertype 63
usrmain 89
utstrcod 106

V

variable description file 28, 53
    attributes 57
    commenting 62
    group information 56
    parameters 54
    sample 53
    scope 56
    structure 54
    unlisted variables 111
variables
    access from compiled routine 95
    accessing through database 95
    dynamic dimensioning 79, 81, 83
    dynamic dimensioning$endrange> . . . 84
    temporary 101
Varname 63, 92
verbose 76