The CÆSAR Code Package
(LA-UR–00-5568, LA-CC–06-027)

Michael L. Hall

January 20, 2010

The CÆSAR Code package was developed by Michael L. Hall. He can be reached at:

Los Alamos National Laboratory
P.O. Box 1663, MS-D413
Los Alamos, NM 87545
Email: `Hall@LANL.gov`

The release history of the CÆSAR Code Package is:

Version: 1.1.1	Date: 05/09/06, 10:51:41 (current version)
Version: 1.1	Date: 03/22/06 (LA-CC–06-027)
Version: 1.0	Date: 02/05/04 (LA-CC–04-009)
Version: 0.6	Date: 10/02/03
Version: 0.5	Date: 11/13/00 (LA-UR–00-5568)

Preface

The Cæsar Code Package is a computational physics development environment. In other words, it provides an environment where the physics of real systems can be modeled, by discretizing a set of partial differential equations on a mesh and solving the resultant algebraic system.

The Cæsar Code Package does not by any means span this extremely large problem space. It does, however, provide a consistent means of incorporating new methods of attacking the computational physics problem. It is *extensible* – new equation sets, new discretizations, new meshes, new linear solvers, new communication libraries, etc., may be incorporated easily.

The emphasis in Cæsar is on equation sets, discretizations, meshes, nonlinear solvers and preconditioners, which are all incorporated into the basic Cæsar structure. In contrast, linear solvers, communications libraries, mesh generators and partitioners, and visualization tools are generally included as external packages developed elsewhere, but may be developed inside Cæsar eventually.

The Cæsar Code Package has these coding characteristics:

- It is written in Fortran-90, preprocessed by Gnu m4.
- It is written in an object-based fashion, and probably comes as close to being object-oriented as is possible in Fortran-90.
- It has both parallel and serial versions, designed in from the start of the project.
- It has a completely levelized design (Lakos, 1996); there are no dependency loops between classes or modules.
- It uses its own form of Design by Contract™(Meyer, 1997) to verify the behavior of all procedures.
- It uses extensive unit testing to certify all classes.
- It uses the ideas of literate programming[1] (Knuth, 1992) to generate documentation (in HTML, PostScript and PDF) from comments included in the code, via the Document Package[2].

The Cæsar Code Package has these computational physics characteristics:

- It allows for multiple mesh types. Currently, a multi-mesh class which can support many types of meshes is being developed. Among the meshes supported will be: uniform meshes, orthogonal meshes, structured meshes, unstructured meshes, adaptive mesh refinement (AMR) meshes, triangular/tetrahedral meshes, and quadrilateral/hexahedral meshes. Polygonal/polyhedral meshes may also be supported in the future.
- It allows for multiple dimensions – 1–D, 2–D and 3–D.
- It allows for multiple geometries – cartesian, cylindrical and spherical.
- It allows for multiple physics packages by allowing for various sets of partial differential equations. Future physics to be modeled may include diffusion, radiation transport (photonics), radiation hydrodynamics, fluid dynamics, magnetohydrodynamics and heat pipe thermal hydraulics.
- It allows for multiple discretizations of the same terms in the equations.
- It allows for multiple external packages for linear solvers, communications, visualization, etc.

[1] `http://www.literateprogramming.com/`
[2] `http://www.lanl.gov/Document`

The Cæsar Code Package is related to the earlier Augustus[3], Spartan[4], and THROHPUT[5] Code Packages.

The documentation is split into the following major parts:

Additional documentation related to the Cæsar project is listed in Presentations and Articles (Chapter 1, page 3).

[3] http://www.lanl.gov/Augustus
[4] http://www.lanl.gov/Spartan
[5] http://www.lanl.gov/THROHPUT

Short Contents

Table of Contents

List of Figures

List of Tables

Part I

Cæsar Package User's Manual

Chapter 1

Presentations and Articles

The papers and presentations in this chapter are available online via the HTML-based version of this document.

Currently, the best general overview papers and presentations concerning the CÆSAR Project are:

Cæsar:

The Cæsar Code: Software Design Issues A presentation by Michael L. Hall that was made to the X-Division External Review Committee on March 10th, 1999 is available in HTML, PDF and PostScript formats (LA-UR–99-1069). There is also a manager's version of this presentation in PDF and PostScript (LA-UR–99-1070).

Spartan:

Spartan/Augustus Overview: Simplified Spherical Harmonics and Diffusion for Unstructured Hexahedral Lagrangian Meshes A presentation by Michael L. Hall that was made to the Shavano working group on April 22nd, 1998 is available in HTML, PDF and PostScript formats (LA-UR–98-3766).

Augustus:

Diffusion Discretization Schemes in Augustus: A New Hexahedral Symmetric Support Operator Method A presentation by Michael L. Hall and Jim E. Morel which was given three times — to the ASCI PI meeting on July 14th, 1998 at Los Alamos National Laboratory; at an X-Division Work In Progress talk on July 29th, 1998; and at the Nuclear Explosives Code Development Conference in Las Vegas, NV on October 29th, 1998 — is available in HTML, PDF and PostScript formats (LA-UR–98-3146).

A Local Support-Operators Diffusion Discretization Scheme for Hexahedral Meshes A paper by J. E. Morel, Michael L. Hall, and Mikhail J. Shashkov which has been submitted to the *Journal of Computational Physics*, Summer 1999 — is available in HTML, PDF and PostScript formats (LA-UR–99-4358).

A Second-Order Cell-Centered Diffusion Difference Scheme for Unstructured Hexahedral Lagrangian Meshes A presentation by Michael L. Hall and Jim E. Morel which was given twice — at the International Congress On Computational And Applied Mathematics in Leuven, Belgium on July 26th, 1996; and at the Nuclear Explosives Code Developers Conference in San Diego, CA on October 24th, 1996 — is available in PostScript and PDF formats (LA-CP–97-7). It was presented with a video.

A Second-Order Cell-Centered Diffusion Difference Scheme for Unstructured Hexahedral Lagrangian Meshes A paper by Michael L. Hall and Jim E. Morel in the *Proceedings of the 1996 Nuclear Explosives Code Developers Conference (NECDC), UCRL-MI-124790* is available in HTML, PDF and PostScript formats (LA-CP–97-8).

A complete listing of presentations and papers that are related to the Cæsar Project follows:

Cæsar:

Progress Towards Higher-Fidelity Yet Efficient Modeling of Radiation Energy Transport Through Three-Dimensional Clouds A poster presentation by Michael L. Hall and Anthony B. Davis that was made at the Atmospheric Radiation Measurement (ARM) Science Team Meeting in Daytona Beach, FL on March 14th–18th, 2005 is available in HTML, PDF and PostScript formats (LA-UR–05-2275).

Three-Dimensional Radiative Transfer, Simplified . . . with Cloud Modeling and Remote Sensing in Mind A presentation by Anthony B. Davis, Michael L. Hall and Igor N. Polonsky that was made at the Atmospheric Radiation Measurement (ARM) Science Team Meeting in Daytona Beach, FL on March 14th–18th, 2005 is available in PDF format (LA-UR–05-2282).

The Cæsar Code: Software Design Issues A presentation by Michael L. Hall that was made to the X-Division External Review Committee on March 10th, 1999 is available in HTML, PDF and PostScript formats (LA-UR–99-1069). There is also a manager's version of this presentation in PDF and PostScript (LA-UR–99-1070).

Spartan:

Diffusion, P_1, and Other Approximate Forms of Radiation Transport A paper by Gordon L. Olson, Larry H. Auer and Michael L. Hall, in the *Journal of Quantitative Spectroscopy and Radiative Transfer*, 64:6 (2000) pp. 619-634, is available in PDF format as both the submitted version and the JQSRT version (LA-UR–99-471).

Analysis of Z Pinch Shock Wave Experiments A Sandia report by Timothy G. Trucano, Kent G. Budge, Jeffery Lawrence, James Asay, Clint Hall, Kathleen Holland, Carl Konrad, Wayne Trott, Gordon Chandler, and Kevin Fleming is available in PDF format (SAND99-1255).

Spartan Test Problem Results A presentation by Michael L. Hall that was made on September 29th, 1998 to the Department of Energy is available in HTML, PDF and PostScript formats (LA-UR–98-3890). It was presented with a video.

Aspects of Analysis of Z Machine Shock Wave Physics Experiments A presentation by Timothy G. Trucano that was made on August 6, 1998 to the DOE Interlaboratory MHD Workshop at Los Alamos National Laboratory is available in PDF format.

Spartan Parallelization and Augustus Time Dependent Results A few slides prepared by Michael L. Hall as part of a larger presentation to Gil Weigand on June 1st, 1998 are available in HTML, PDF and PostScript formats (LA-UR–98-3750).

Spartan/Augustus Overview: Simplified Spherical Harmonics and Diffusion for Unstructured Hexahedral Lagrangian Meshes A presentation by Michael L. Hall that was made to the Shavano working group on April 22nd, 1998 is available in HTML, PDF and PostScript formats (LA-UR–98-3766).

"Top Hat" problem (or "Pipe Flow" problem) Results From Spartan Some results and graphs by Michael L. Hall from running Spartan on the "Top Hat" problem are available: Data Listing, Results Summary (PostScript) and compressed tarfile of whole directory.

Augustus:

A Local Support Operator Diffusion Discretization Scheme for Hexahedral Meshes A presentation by Michael L. Hall, Jim E. Morel and Mikhail J. Shashkov which was given at the JOWOG 42 meeting on October 21st, 1999 at Los Alamos National Laboratory — is available in HTML, PDF and PostScript formats (LA-UR–99-5834).

A Local Support-Operators Diffusion Discretization Scheme for Hexahedral Meshes A presentation by Jim Morel, Michael L. Hall, and Mikhail J. Shashkov which was presented at a CNLS Numerical Analysis Seminar on November 9th, 1999 at Los Alamos National Laboratory — is available in HTML, PDF and PostScript formats (LA-UR–99-6259).

A Local Support-Operators Diffusion Discretization Scheme for Hexahedral Meshes A paper by J. E. Morel, Michael L. Hall, and Mikhail J. Shashkov which has been submitted to the *Journal of Computational Physics*, Summer 1999 — is available in HTML, PDF and PostScript formats (LA-UR–99-4358).

Diffusion Discretization Schemes in Augustus: A New Hexahedral Symmetric Support Operator Method A presentation by Michael L. Hall and Jim E. Morel which was given three times — to the ASCI PI meeting on July 14th, 1998 at Los Alamos National Laboratory; at an X-Division Work In Progress talk on July 29th, 1998; and at the Nuclear Explosives Code Development Conference in Las Vegas, NV on October 29th, 1998 — is available in HTML, PDF and PostScript formats (LA-UR–98-3146).

A Second-Order Cell-Centered Diffusion Difference Scheme for Unstructured Hexahedral Lagrangian Meshes A presentation by Michael L. Hall and Jim E. Morel which was given twice — at the International Congress On Computational And Applied Mathematics in Leuven, Belgium on July 26th, 1996; and at the Nuclear Explosives Code Developers Conference in San Diego, CA on October 24th, 1996 — is available in PostScript and PDF formats (LA-CP–97-7). It was presented with a video.

A Second-Order Cell-Centered Diffusion Difference Scheme for Unstructured Hexahedral Lagrangian Meshes A paper by Michael L. Hall and Jim E. Morel in the *Proceedings of the 1996 Nuclear Explosives Code Developers Conference (NECDC), UCRL-MI-124790* is available in HTML, PDF and PostScript formats (LA-CP–97-8).

Unique Linear Solver Needs of the Los Alamos Radiation Transport Team A presentation by Michael L. Hall and John M. McGhee which discusses the peripheral issue of linear solvers with respect to two main code packages, DANTE and Augustus/Spartan, is available in PostScript and PDF formats (LA-UR–99-1225). It was presented at the Application Code Developers Meeting of the Accelerated Strategic Computing Initiative in Albuquerque, NM, August 28th, 1996.

RAYHYD: An ICF Target Simulation Code Written in C++ A paper by Kent G. Budge, James S. Peery, Timothy G. Trucano, Mike K. Wong, Jim E. Morel and Michael L. Hall in the *Proceedings of the 1994 Nuclear Explosives Code Developers Conference (NECDC)* is available in PDF format.

Augustus: Unstructured Quadrilateral Mesh Diffusion A preliminary results presentation by Michael L. Hall and Jim E. Morel that was made very early in the development of Augustus, without much detail, is available in PostScript and PDF. It was presented internally at Sandia National Laboratory on May 6th, 1994.

Chapter 2

Installation

2.1 Requirements

In order to unpack the distribution file, you must have:

GNU tar[1] available and in your path. CÆSAR has recently been tested with gtar version 1.13.25.

GNU zip[2] available and in your path. CÆSAR has recently been tested with gzip version 1.3.3.

In order to build and run the program, you must also have:

GNU make[3] available and in your path. CÆSAR has recently been tested with gmake version 3.79.1.

GNU m4[4] available and in your path. CÆSAR has recently been tested with gm4 version 1.4.1.

A Fortran 90/95 compiler available and in your path. These compilers are currently supported:

Lahey Compiler on Linux.[5] CÆSAR has recently been tested with Lahey F95 version L6.20c.

Absoft Compiler on Linux.[6] CÆSAR has recently been tested with Absoft F95 version 8.2.

Absoft Compiler on Darwin.[7] CÆSAR has recently been tested with Absoft F95 version 8.2.

Sun Workshop Compiler on Solaris.[8] CÆSAR has recently been tested with Sun WorkShop 6 update 2 Fortran 95 6.2 on Solaris 6. (Yes, this is very old. Support will soon be removed from Solaris.)

These compilers have been supported in the past:

MIPSpro Compiler on SGI/IRIX.[9] CÆSAR was last tested with MIPSpro compiler version 7.4 on IRIX 6.5.

In order to generate the documentation yourself, you must also have:

[1] http://www.gnu.org/software/tar/
[2] http://www.gzip.org/
[3] http://www.gnu.org/software/make/
[4] http://www.gnu.org/software/m4/
[5] http://www.lahey.com/
[6] http://www.absoft.com/
[7] http://www.absoft.com/
[8] http://www.sun.com/
[9] http://www.sgi.com/products/software/irix/tools/mipspro.html

The Document Package[10] available and in your path.

2html[11] installed on your system. The latex2html-2002.2.1-6.rpm version is sufficient.

perl[12] installed on your system and in your path. Cæsar has recently been tested with perl version 5.8.0.

Alternately, you can retrieve the documentation from the web[13].

2.2 External Packages

Cæsar makes use of several external packages to accomplish its goals.

In most cases, there are (or will be) multiple packages to perform any given function, so that Cæsar may be compiled in many different configurations. For selected packages, optimization and debugging can be toggled as a group.

2.2.1 MPI Package

MPI, the Message Passing Interface standard, was developed by the MPI Forum to provide a standard for low-level interprocessor communication in a parallel environment. Several vendors have implementations of MPI and there is also a portable version of MPI called MPICH.

Cæsar does not require MPI directly, but all parallel versions of Cæsar use packages which require MPI. Cæsar can be compiled with no MPI (a serial version), or with a vendor-supplied MPI, or with the included MPICH.

2.2.2 PGSLib Package

PGSLib(Parallel Gather-Scatter Library) was developed by Robert Ferrell of Cambridge Power Computing Associates, Ltd.. It is used for high-level parallel communication in Cæsar. PGSLib has both a serial version, which runs on a single processor, and a parallel version, which allows inter-processor communication.

The parallel version of PGSLib requires an MPI package (see Section 2.2.1), which can be provided by the computer vendor or by MPICH (included with Cæsar).

Cæsar can be compiled without PGSLib, or with the serial or parallel versions of PGSLib.

2.2.3 LAPACK Package

LAPACK (Linear Algebra Package) provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS). LAPACK was developed by Jack Dongarra's LAPACK Project group at the University of Tennessee.

Highly efficient machine-specific implementations of LAPACK and the BLAS are available for many modern high-performance computers. If a vendor-supplied version is available, it is prefered.

[10] http://www.lanl.gov/Document
[11] http://tug.org/mailman/listinfo/latex2html
[12] http://www.perl.org/
[13] `http://www.lanl.gov/Caesar/`

Cæsar does not currently use LAPACK directly, but it may in the future. Cæsar currently requires the LAMG package, which may be compiled with LAPACK. Cæsar can be compiled with no LAPACK or with a vendor-supplied LAPACK, or with the included LAPACK.

2.2.4 LAMG Package

LAMG (Los Alamos Algebraic Multigrid Code) was developed by the Parallel Architectures and Algorithms Team in the Scientific Computing Group (CIC-19) at LANL. It provides algebraic multigrid and Krylov solvers to the Cæsar Project.

Cæsar can not currently be compiled without LAMG because LAMG provides the only solvers in Cæsar. Eventually, as other solver packages are added and developed within Cæsar, this dependency will be removed.

Chapter 3

Standalone Usage

3.1 Graphical User Interface

This section will be expanded in the future.

Chapter 4

Calling the Cæsar Package

This section will be expanded in the future.

Part II

Cæsar Package Code Manual

Chapter 5

Design Decisions

- Documentation will be an integral part of the program, and will be done via my Document package. The documentation will be available in both HTML and PostScript form, via LaTeX and LaTeX2html.
- Assume `gmake` is available. This allows many coding constructions not found in vendor `make`s, and increases portability.
- "Design by Contract" coding methodology will be used, but some terminology may be changed to be more meaningful.
  - For instance, "assertions" will be used, but will be referred to as "verifications". To "assert" means to state that something is true, whereas to "verify" means to test to make sure that something is true. Other nomenclature that I considered and rejected includes: check, confirm, certify, warrant, ascertain, ensure, guarantee. All of these are better than "assert", but I considered "verify" to be the best.
  - Also, "invariant" will be referred to as "valid_state", because it is more indicative of the test being done. Other nomenclature that I considered and rejected includes: self-consistent, consistent, well-formed, valid.

  Granted, I may lose a small amount of understanding on the part of C programmers who are looking at my code (and are familiar with the terms "assert" and "invariant"), but I will gain a great deal of understanding from people that are new to the "Design by Contract" method.
- Assume `gm4` is available. This means that line numbers from the source code can be output as error messages from the verifications.
- Assume that all error line references are to the original, unprocessed files. This means that any header files can add blank lines, which means that the ubiquitous "dnl" can be eliminated, which will greatly increase readability.
- In addition to the VERIFY macros, I'm making a WARN_IF macro that generates a warning if a test fails but continues program execution.
- I considered using DejaGnu to automate testing, but ultimately rejected it. The reasons for rejection include:
  - It seems that DejaGnu's main thrust is testing on different platforms, and it requires Tcl and Expect to run. I don't want to require all three programs (Tcl, Expect and DejaGnu) to be able to test on other architectures.
  - It looks non-trivial to set up, especially since I don't know Tcl and Expect.
  - It's unclear what benefit, if any, would be derived from using it.

My current solution for testing, as far as modules are concerned, is to include a test routine at the end of each module which will only be activated if a gm4 flag is set. I'm including the "script" necessary to run each module test as Self-Documentation text that my perl script Document can pick out.

Chapter 6

m4 Preprocessing

The CÆSAR Code Package makes extensive use of the m4 preprocessor to modify the Fortran 90 source code. All Fortran 90 source is preprocessed by m4 before compilation. The Gnu version of m4 is required.

Some of the uses of the m4 preprocessor in CÆSAR include unit testing, verification and warning statements (which implement design by contract), intrinsic type definition, and code replication.

6.1 Global m4 Settings

The "settings" set of m4 commands defines the m4 environment that is used in the CÆSAR Code Package:

- The m4 quotation characters are changed to "[" and "]", to avoid interaction with standard F90 syntax.
- The m4 comment character is changed to "!", to coincide with the F90 comment character. Note that comments in m4 are echoed to the output with no macro expansion. To avoid echoing at all, the m4 command "dnl" must be used (this may change in future versions of Gnu m4).
- Builtin m4 macros must be called with a prefix of "m4_". For example, the "format" macro must be called via "m4_format". Therefore, all of the m4 preprocessing in the CÆSAR Code Package must be done with Gnu m4, using either the `-P` or `--prefix-builtins` option.
- Certain commonly used m4 builtins are defined to be usable without an "m4_" prefix.

In addition to these environmental settings, some globally useful macros are defined.

m4 macros defined in the `include/settings.m4` file:

`__date__`	Date field (mm/dd/yy).
`__file__`	Unprefixed form of `m4___file__`.
`__line__`	Unprefixed form of `m4___line__`.
`__time__`	Time field (hh:mm:ss, 24-hour).
`define`	Unprefixed form of `m4_define`.
`dnl`	Unprefixed form of `m4_dnl`.
`expand`	Force macro expansion in words containing underscores.
`firstword`	Returns the first word from a space-delimited list of words.
`forloop`	A numerical text iterator (see code listing in section A.1 for input/output details).
`fortext`	A textual text iterator (see code listing in section A.1 for input/output details).
`ifdef`	Unprefixed form of `m4_ifdef`.
`ifelse`	Unprefixed form of `m4_ifelse`.

`include`	Unprefixed form of `m4_include`.
`m4_chop`	Removes last character of input string.
`m4_die`	Prints error message and terminates.
`popdef`	Unprefixed form of `m4_popdef`.
`pushdef`	Unprefixed form of `m4_pushdef`.
`tailwords`	Returns everything except the first word from a space-delimited list of words.
`undefine`	Unprefixed form of `m4_undefine`.

The Settings m4 Macros code listing in section A.1 contains additional documentation.

6.2 Type m4 Macros

The "type" set of m4 macros allows intrinsic F90 types to be defined in a manner similar to defined types, making use of the kind parameter to ensure consistency and allow for double and single precision versions of the code. The type macros allow the following constructions to be used in the CÆSAR Code Package:

```
type(real) :: realvar1
type(real,0) :: realvar2
type(real,3) :: realvar3
type(integer,1), intent(out) :: intvar1
type(integer,4,np) :: intvar2
type(logical,1,np) :: logvar
type(character,8) :: charvar1
type(character,*,2,np) :: charvar2
type(defined) :: defvar

intvar = changetype(integer, variable)
realvar = changetype(real, variable)
```

This code is expanded by Gnu m4 into the following valid F90 code:

```
real (kind=KIND(1.0d0)) :: realvar1
real (kind=KIND(1.0d0)) :: realvar2
real (kind=KIND(1.0d0)), pointer, dimension(:,:,:) :: realvar3
integer (kind=KIND(1)), pointer, dimension(:), intent(out) :: intvar1
integer (kind=KIND(1)), dimension(:,:,:,:) :: intvar2
logical (kind=KIND(.true.)), dimension(:) :: logvar
character (len=8) :: charvar1
character (len=*), dimension(:,:) :: charvar2
type(defined) :: defvar

intvar = INT(variable, KIND(1))
realvar = REAL(variable, KIND(1.0d0))
```

Note that this set of m4 macros depends on the m4 commands in the `settings.m4` file (see section 6.1) and on the SINGLE and UNICOS macro definitions.

m4 macros defined in the `include/types.m4` file:

`changetype`	Used for intrinsic type conversions.
`pnt$dim`	Private macro used in type.
`real$kind`	Private macro used in type and changetype.
`type`	Used for intrinsic type definition. (see code listing in section A.2 for input/output details).

The Type m4 Macros code listing in section A.2 contains additional documentation.

6.3 Verify m4 Macros

> *Power is not revealed by striking hard or often, but by striking true. – Honore de Balzac (1799-1850)*

The "verify" set of m4 macros enable the conditional compilation of the verification statements which are used to implement "Design by Contract" methodology in the CÆSAR Code Package. Verification statements[1] are simply logical tests on variables which can be conditionally compiled into the code, allowing for extreme error checking if they are compiled in and unfettered execution speed if they are compiled out. The "Design by Contract" methodology extends this idea slightly by specifying when these verifications are to be done: input requirements are verified upon entry into a routine and output guarantees are verified before routine exit.

The CÆSAR Code Package further extends this idea in two ways:

- In addition to VERIFY commands which *halt* execution if they are not satisfied, WARN_IF commands output a warning message and *continue* execution if they are not satisfied.
- Both VERIFY and WARN_IF commands are conditionally compiled in based on a level variable (`DEBUG_LEVEL` for VERIFY and `WARNING_LEVEL` for WARN_IF) which can be set for each instance of a command individually. This allows a great deal of user control.

The syntax for the VERIFY/WARN_IF macros is given by:

```
VERIFY(<logical expression>,<activation level>)
WARN_IF(<logical expression>,<activation level>)
```

where `<logical expression>` is the F90 test to be satisfied and `<activation level>` is the value of the appropriate LEVEL variable (`DEBUG_LEVEL` for VERIFY and `WARNING_LEVEL` for WARN_IF) at which the test is compiled into the executable program – for level values less than the `<activation level>`, the test is commented out. To reiterate, if the debug or warning level is not set high enough to activate a particular command, that command will be commented out and will not be executed by the code.

The level at which a particular command will activate should be set according to a combination of criteria: importance and cost. An expensive (in terms of cpu-time) command, ceteris paribus, should have a higher `<activation level>` than a cheaper command. Commands whose satisfaction is more important should have a lower `<activation level>`. Verification or warning commands that specify an `<activation level>` of -1 will always be executed.

Code compiled with lower debug or warning levels will necessarily be as fast as or faster than code compiled with higher levels, but will also include fewer checks for accuracy. A debug or warning level of zero should include only those checks that the code author thought were absolutely necessary. Setting the debug or warning level to -1 will eliminate all commands of the specified type (except those with activation levels of -1).

When a VERIFY/WARN_IF test is compiled into the code, an if-block is inserted which evaluates the test. If the test fails, an output message is printed which contains the text, filename, and line number of the failed test. VERIFY commands then abort program execution, while WARN_IF commands continue.

There is a slight difference in the coding for the Intrinsic Classes (see chapter 7), because they are lower level than the communication routines, and therefore require serial coding. To trigger this difference, the Intrinsic Classes specify "`define([VERIFY_COMMUNICATION],[Local])`" in their headers.

[1] C and C++ programmers call a similar concept "assertions", see B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 1988, page 253

As an example, assume that a file named `verify_example` contains these lines:

```
m4_include(settings.m4)
m4_include(verify.m4)

VERIFY( i < 1, 5)
VERIFY( j > 2, 6)
VERIFY( k < 4, 7)

define([DEBUG_LEVEL], 2)
VERIFY( Valid_State(matrix), 1)
VERIFY( Valid_State(matrix), 2)
VERIFY( Valid_State(matrix), 3)
```

When this file is processed by Gnu m4, it is expanded into the following valid F90 code, if the `DEBUG_LEVEL` variable is not set:

```
! if (.not. Global_ALL(i < 1)) then
!   if (this_is_IO_PE) then
!     write (6,*) "Verification failed: ", &
!                 "i < 1, ", &
!                 "file verify_example, ", &
!                 "line 4."
!   end if
!   call Abort
! end if
! if (.not. Global_ALL(j > 2)) then
!   if (this_is_IO_PE) then
!     write (6,*) "Verification failed: ", &
!                 "j > 2, ", &
!                 "file verify_example, ", &
!                 "line 5."
!   end if
!   call Abort
! end if
! if (.not. Global_ALL(k < 4)) then
!   if (this_is_IO_PE) then
!     write (6,*) "Verification failed: ", &
!                 "k < 4, ", &
!                 "file verify_example, ", &
!                 "line 6."
!   end if
!   call Abort
! end if

 if (.not. Global_ALL(Valid_State(matrix))) then
   if (this_is_IO_PE) then
     write (6,*) "Verification failed: ", &
                 "Valid_State(matrix), ", &
                 "file verify_example, ", &
                 "line 9."
   end if
   call Abort
 end if
 if (.not. Global_ALL(Valid_State(matrix))) then
```

```
      if (this_is_IO_PE) then
        write (6,*) "Verification failed: ", &
                    "Valid_State(matrix), ", &
                    "file verify_example, ", &
                    "line 10."
      end if
      call Abort
    end if
   ! if (.not. Global_ALL(Valid_State(matrix))) then
   !   if (this_is_IO_PE) then
   !     write (6,*) "Verification failed: ", &
   !                 "Valid_State(matrix), ", &
   !                 "file verify_example, ", &
   !                 "line 11."
   !   end if
   !   call Abort
   ! end if
```

If the DEBUG_LEVEL m4 variable is set to a value of 6, via the command line option -DDEBUG_LEVEL=6, then the expansion becomes:

```
    if (.not. Global_ALL(i < 1)) then
      if (this_is_IO_PE) then
        write (6,*) "Verification failed: ", &
                    "i < 1, ", &
                    "file verify_example, ", &
                    "line 4."
      end if
      call Abort
    end if
    if (.not. Global_ALL(j > 2)) then
      if (this_is_IO_PE) then
        write (6,*) "Verification failed: ", &
                    "j > 2, ", &
                    "file verify_example, ", &
                    "line 5."
      end if
      call Abort
    end if
   ! if (.not. Global_ALL(k < 4)) then
   !   if (this_is_IO_PE) then
   !     write (6,*) "Verification failed: ", &
   !                 "k < 4, ", &
   !                 "file verify_example, ", &
   !                 "line 6."
   !   end if
   !   call Abort
   ! end if

    if (.not. Global_ALL(Valid_State(matrix))) then
      if (this_is_IO_PE) then
        write (6,*) "Verification failed: ", &
                    "Valid_State(matrix), ", &
                    "file verify_example, ", &
                    "line 9."
```

```
      end if
      call Abort
    end if
    if (.not. Global_ALL(Valid_State(matrix))) then
      if (this_is_IO_PE) then
        write (6,*) "Verification failed: ", &
                    "Valid_State(matrix), ", &
                    "file verify_example, ", &
                    "line 10."
      end if
      call Abort
    end if
  ! if (.not. Global_ALL(Valid_State(matrix))) then
  !   if (this_is_IO_PE) then
  !     write (6,*) "Verification failed: ", &
  !                 "Valid_State(matrix), ", &
  !                 "file verify_example, ", &
  !                 "line 11."
  !   end if
  !   call Abort
  ! end if
```

If the WARN_IF macro had been used instead of the VERIFY macro (and the `WARNING_LEVEL` were set instead of the `DEBUG_LEVEL`), then the only changes in the output of the above examples would be that the Abort call would not have appeared, the test would not have been negated, and a Global_ANY would have been used instead of a Global_ALL.

If the VERIFY_COMMUNICATION macro had been set to Local, then the Global_ALL call would not have been used, the IO_PE check would not have appeared, and the Abort call would have been replaced with a stop.

As an additional refinement, any quotation characters in the logical expression are removed from the output string to prevent confusion with quoting there. Quotation marks in the logical expression should therefore be paired.

Note that this set of m4 macros depends on the m4 commands in the `settings.m4` file (see section 6.1) and on the DEBUG_LEVEL and WARNING_LEVEL macro definitions.

m4 macros defined in the `include/verify.m4` file:

`DEBUG_LEVEL`	Verification is turned on for VERIFY commands whose activation level is equal to or less than the `DEBUG_LEVEL`.
`VERIFY`	Compile in or comment out a test which halts execution if unsatisfied.
`WARNING_LEVEL`	Warnings are turned on for WARN_IF commands whose activation level is equal to or less than the `WARNING_LEVEL`.
`WARN_IF`	Compile in or comment out a test which prints a warning and continues execution if satisfied.

The Verify m4 Macros code listing in section A.3 contains additional documentation.

6.4 Replicate m4 Macros

The "replicate" set of m4 macros is used to replicate a routine so that a version exists for every possible array dimensioning (scalars plus up to seven-dimensioned arrays, the maximum allowed in F90). To use the replicate macros in the Cæsar Code Package,

- define the interface using the `REPLICATE_INTERFACE` macro,
- define the `REPLICATE_ROUTINE`, using the `REP_EXPAND`, `ARRAY_ONLY` and `SCALAR_ONLY` macros as needed,
- invoke the `REPLICATE` macro to generate multiple versions of the `REPLICATE_ROUTINE` for scalars and all array dimensions, or
- invoke the `REPLICATE_ARRAYS` macro to generate multiple versions of the `REPLICATE_ROUTINE` for all array dimensions only, and write a separate routine for scalars (sometimes useful if the array and scalar versions have little in common).

Take the following code segment as an example:

```
module test_replicate
  REPLICATE_INTERFACE([Generic_Routine], [Specific_Routine])
contains

define([REPLICATE_ROUTINE],[
  function Specific_Routine_$1 (R REP_ARGS([var[]i]))
    type(real,$1) :: R
    REP_DECLARE([type(integer)], [var[]i])
    REP_ALLOCATE(R, [var[]i], [status])
    ARRAY_ONLY DEALLOCATE(R)
    SCALAR_ONLY R = 999.
    <other routine contents>
  end function Specific_Routine_$1

])
REPLICATE
end module test_replicate
```

This code is expanded by Gnu m4 into the following valid F90 code:

```
module test_replicate
  interface Generic_Routine
    module procedure Specific_Routine_0
    module procedure Specific_Routine_1
    module procedure Specific_Routine_2
    module procedure Specific_Routine_3
    module procedure Specific_Routine_4
    module procedure Specific_Routine_5
    module procedure Specific_Routine_6
    module procedure Specific_Routine_7
  end interface
contains

  function Specific_Routine_0 (R)
    real (kind=KIND(1.0d0)) :: R
    ! DEALLOCATE(R)
    R = 999.
    <other routine contents>
  end function Specific_Routine_0

  function Specific_Routine_1 (R, var1)
    real (kind=KIND(1.0d0)), pointer, dimension(:) :: R
    integer (kind=KIND(1)) :: var1
```

```
    ALLOCATE(R(var1), stat=status)
    DEALLOCATE(R)
    ! R = 999.
    <other routine contents>
  end function Specific_Routine_1

  function Specific_Routine_2 (R, var1, var2)
    real (kind=KIND(1.0d0)), pointer, dimension(:,:) :: R
    integer (kind=KIND(1)) :: var1, var2
    ALLOCATE(R(var1, var2), stat=status)
    DEALLOCATE(R)
    ! R = 999.
    <other routine contents>
  end function Specific_Routine_2

  function Specific_Routine_3 (R, var1, var2, var3)
    real (kind=KIND(1.0d0)), pointer, dimension(:,:,:) :: R
    integer (kind=KIND(1)) :: var1, var2, var3
    ALLOCATE(R(var1, var2, var3), stat=status)
    DEALLOCATE(R)
    ! R = 999.
    <other routine contents>
  end function Specific_Routine_3

  function Specific_Routine_4 (R, var1, var2, var3, var4)
    real (kind=KIND(1.0d0)), pointer, dimension(:,:,:,:) :: R
    integer (kind=KIND(1)) :: var1, var2, var3, var4
    ALLOCATE(R(var1, var2, var3, var4), stat=status)
    DEALLOCATE(R)
    ! R = 999.
    <other routine contents>
  end function Specific_Routine_4

  function Specific_Routine_5 (R, var1, var2, var3, var4, var5)
    real (kind=KIND(1.0d0)), pointer, dimension(:,:,:,:,:) :: R
    integer (kind=KIND(1)) :: var1, var2, var3, var4, var5
    ALLOCATE(R(var1, var2, var3, var4, var5), stat=status)
    DEALLOCATE(R)
    ! R = 999.
    <other routine contents>
  end function Specific_Routine_5

  function Specific_Routine_6 (R, var1, var2, var3, var4, var5, var6)
    real (kind=KIND(1.0d0)), pointer, dimension(:,:,:,:,:,:) :: R
    integer (kind=KIND(1)) :: var1, var2, var3, var4, var5, var6
    ALLOCATE(R(var1, var2, var3, var4, var5, var6), stat=status)
    DEALLOCATE(R)
    ! R = 999.
    <other routine contents>
  end function Specific_Routine_6

  function Specific_Routine_7 (R, var1, var2, var3, var4, var5, var6, var7)
    real (kind=KIND(1.0d0)), pointer, dimension(:,:,:,:,:,:,:) :: R
    integer (kind=KIND(1)) :: var1, var2, var3, var4, var5, var6, var7
```

```
    ALLOCATE(R(var1, var2, var3, var4, var5, var6, var7), stat=status)
    DEALLOCATE(R)
    ! R = 999.
    <other routine contents>
  end function Specific_Routine_7

end module test_replicate
```

Note that this set of m4 macros depends on the m4 commands in the `settings.m4` file (see section 6.1) and on the `REPLICATE_ROUTINE` macro definition.

m4 macros defined in the `include/replicate.m4` file:

`ARRAY_ONLY`	Used to comment out lines for scalars, but leave them untouched for arrays.
`REPLICATE`	Replicates a routine for all array dimensions and scalars.
`REPLICATE_ARRAYS`	Replicates a routine for all array dimensions only.
`REPLICATE_INTERFACE`	Defines the interface block for the replicated routine.
`REP_ALLOCATE`	Used to allocate a replicated variable.
`REP_ARGS`	Used to replicate an argument list.
`REP_DECLARE`	Used to decalre a replicated variable.
`REP_EXPAND`	Used in the `REPLICATE_ROUTINE` macro to expand to text that varies with the dimensioning (see code listing in section A.4 for input/output details).
`REP_NUMBER`	The number of the dimension of the current routine, used internally.
`SCALAR_ONLY`	Used to comment out lines for arrays, but leave them untouched for scalars.
`type(vartype,$1)`	A useful construction for replicated procedures which is defined by the types macros (see section 6.2).

The Replicate m4 Macros code listing in section A.4 contains additional documentation.

6.5 Superclass m4 Macros

The "superclass" set of m4 macros is used to automatically create a superclass module that will dynamically dispatch superclass calls to the correct subclass routine or function. This achieves run-time polymorphism, which is commonly considered to be too much effort in Fortran 90, in an easily implemented fashion.

To use the "superclass" set of m4 macros certain naming and coding conventions must be followed. Assuming that the superclass name is "Matrix", that three subclasses exist named "One", "Two", and "Three", and that the routine to be dynamically dispatched is named "Solve", the following names must be used:

superclass derived type:	Matrix_type
subclass derived types:	One_type, Two_type, Three_type
subclass module names:	One_Class, Two_Class, Three_Class
superclass module name:	Matrix_Class
superclass routine name:	Solve_Matrix
superclass interface name:	Solve
subclass routine names:	Solve_One, Solve_Two, Solve_Three
subclass interface name:	Solve

In addition, the superclass derived type must be:

```
  type Matrix_type
```

```
    character (len=80) :: Subclass
    type(One_type) :: One
    type(Two_type) :: Two
    type(Three_type) :: Three
  end type Matrix_type
```

To use the superclass macros in the Cæsar Code Package,

- define the superclass name in the `SUPERCLASS` macro,
- define the subclass names as a space-delimited list in the `SUBCLASSES` macro,
- use the `SUPERCLASS_USE_ASSOCIATIONS`, `SUPERCLASS_TYPE`, `SUPERCLASS_ROUTINE` and `SUPERCLASS_-FUNCTION` macros as needed to define the superclass module.

Take the following code segment as an example:

```
define([SUPERCLASS],[Matrix])
define([SUBCLASSES],[One Two Three])

module SUPERCLASS[]_Class

  SUPERCLASS_USE_ASSOCIATIONS
  SUPERCLASS_TYPE

contains

  SUPERCLASS_ROUTINE([Initialize],
                     [type(real)], [a], [The a variable],
                     [type(integer), intent(in)], [b], [The b variable])

  SUPERCLASS_FUNCTION([Verify_State], [type(logical)],
                      [type(real)], [b], [The b variable])

  SUPERCLASS_ROUTINE([Finalize],
                     [type(real)], [c], [The c variable],
                     [type(real)], [d], [The d variable])

end module SUPERCLASS[]_Class
```

This code is expanded by Gnu m4 into the following valid F90 code:

```
module Matrix_Class

  use One_Class
  use Two_Class
  use Three_Class

  type Matrix_type
    character (len=80) :: Subclass
    type(One_type) :: One
    type(Two_type) :: Two
    type(Three_type) :: Three
  end type Matrix_type

contains
```

```
subroutine Initialize_Matrix (Matrix, a, b)

  type(Matrix_type) Matrix
  real (kind=KIND(1.0d0)) :: a ! The a variable
  integer (kind=KIND(1)), intent(in) :: b ! The b variable

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  select case (Matrix%Subclass)
  case ("One")
    call Initialize (Matrix%One, a, b)
  case ("Two")
    call Initialize (Matrix%Two, a, b)
  case ("Three")
    call Initialize (Matrix%Three, a, b)
  case default
    write (6,*) 'Error: no ', Matrix%Subclass, ' in Matrix_Class.'
  end select

end subroutine Initialize_Matrix

function Verify_State_Matrix (Matrix, b)

  type(Matrix_type) Matrix
  logical (kind=KIND(.true.)) :: Verify_State, Verify_State_Matrix
  real (kind=KIND(1.0d0)) :: b ! The b variable

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  select case (Matrix%Subclass)
  case ("One")
    Verify_State_Matrix = Verify_State (Matrix%One, b)
  case ("Two")
    Verify_State_Matrix = Verify_State (Matrix%Two, b)
  case ("Three")
    Verify_State_Matrix = Verify_State (Matrix%Three, b)
  case default
    write (6,*) 'Error: no ', Matrix%Subclass, ' in Matrix_Class.'
  end select

end function Verify_State_Matrix

subroutine Finalize_Matrix (Matrix, c, d)

  type(Matrix_type) Matrix
  real (kind=KIND(1.0d0)) :: c ! The c variable
  real (kind=KIND(1.0d0)) :: d ! The d variable

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  select case (Matrix%Subclass)
  case ("One")
    call Finalize (Matrix%One, c, d)
```

```
    case ("Two")
      call Finalize (Matrix%Two, c, d)
    case ("Three")
      call Finalize (Matrix%Three, c, d)
    case default
      write (6,*) 'Error: no ', Matrix%Subclass, ' in Matrix_Class.'
    end select

  end subroutine Finalize_Matrix

end module Matrix_Class
```

Note that this set of m4 macros depends on the m4 commands in the `settings.m4` file (see section 6.1) and on the `SUPERCLASS` and `SUBCLASSES` macro definitions.

m4 macros defined in the `include/superclass.m4` file:

`SUPERCLASS_ARGUMENTS`	Used internally by the `SUPERCLASS_ROUTINE` and `SUPERCLASS_FUNCTION` macros.
`SUPERCLASS_DECLARATIONS`	Used internally by the `SUPERCLASS_ROUTINE` and `SUPERCLASS_FUNCTION` macros.
`SUPERCLASS_FUNCTION`	Expands into a complete function for the superclass. This function dynamically dispatches calls to the superclass to the correct subclass function.
`SUPERCLASS_ROUTINE`	Expands into a complete subroutine for the superclass. This subroutine dynamically dispatches calls to the superclass to the correct subclass routine.
`SUPERCLASS_TYPE`	Outputs a standard superclass type definition.
`SUPERCLASS_USE_ASSOCIATIONS`	Outputs the correct use associations for the superclass.

The Superclass m4 Macros code listing in section A.5 contains additional documentation.

6.6 Unit Test m4 Macros

> *We like to test things... No matter how good an idea sounds, test it first. – Henry Block, CEO, H&R Block*

The "unit test" set of m4 macros provides the capability to conditionally compile in F90 statements for use during unit testing. Unit testing is the ability to test a particular routine in isolation from the remainder of the code. Unit testing is accomplished by including a main program and necessary auxiliary routines at the end of the file containing the routine to be tested, in this manner:

```
ifdef([UNIT_TEST],[
program Unit_Test
  <program contents>
end
])
```

During a standard compilation, the Unit_Test program will not be present because UNIT_TEST is not defined. To do unit testing, the file is processed with "-DUNIT_TEST" defined on the m4 command line, and the Unit_Test program is compiled in.

In addition to those capabilities, which are provided in each F90 source file, the "unit test" set of m4 macros allows the following constructions to be used in the Cæsar Code Package:

```
TESTWRITE (6,100) 'Output test variables ==>', &
  IF_UNIT_TEST x1, x2, x3
IF_UNIT_TEST 100 format (a, 3(1pe13.6))
IF_NOT_UNIT_TEST debug = 0
```

This code is expanded by Gnu m4 into the following valid F90 code:

```
! write (6,100) 'Output test variables ==>', &
  ! x1, x2, x3
! 100 format (a, 3(1pe13.6))
debug = 0
```

if UNIT_TEST is not defined. If UNIT_TEST is defined, then the same code is expanded by Gnu m4 into the following:

```
write (6,100) 'Output test variables ==>', &
  x1, x2, x3
100 format (a, 3(1pe13.6))
! debug = 0
```

which allows for statements to be conditionally compiled in when unit testing is being done.

Note that this set of m4 macros depends on the m4 commands in the `settings.m4` file (see section 6.1) and on the UNIT_TEST macro definition.

m4 macros defined in the `include/unit_test.m4` file:

`TESTWRITE`	Write statement toggled by the UNIT_TEST macro.
`IF_UNIT_TEST`	Comments out code unless UNIT_TEST is defined.
`IF_NOT_UNIT_TEST`	Comments out code if UNIT_TEST is defined.

The Unit Test m4 Macros code listing in section A.6 contains additional documentation.

6.7 Flags Module

The Flags Module is used to define flag constants that are used in the CÆSAR Code Package.

Flags Module public parameters:

Intrinsic Initialization/Finalization Flags	
`finalize_character_flag`	Value used to finalize characters.
`finalize_integer_flag`	Value used to finalize integers.
`finalize_logical_flag`	Value used to finalize logicals.
`finalize_real_flag`	Value used to finalize reals.
`initialize_character_flag`	Value used to initialize characters.
`initialize_integer_flag`	Value used to initialize integers.
`initialize_logical_flag`	Value used to initialize logicals.
`initialize_real_flag`	Value used to initialize reals.
Derived Type Initialization Flags	
`initialized_flag`	Value used to signify derived type initialization.
`uninitialized_flag`	Value used to signify derived type lack of initialization (when a derived type is finalized).
Boundary Condition Face Flags	

`AMR_Large_Cell_BC`	In an AMR mesh, this signifies an internal level-jump interface where the current cell is the larger cell.
`AMR_Small_Cell_BC`	In an AMR mesh, this signifies an internal level-jump interface where the current cell is the smaller cell.
`Dirichlet_BC`	Signifies a Dirichlet boundary condition.
`Homogeneous_BC`	Signifies a homogeneous boundary condition.
`Internal_or_Periodic_BC`	Signifies an internal face or a periodic boundary condition face.
`Neumann_BC`	Signifies a Neumann boundary condition.
`Reflective_BC`	Signifies a reflective boundary condition.
`Source_BC`	Signifies a source boundary condition.
`Vacuum_BC`	Signifies a vacuum boundary condition.

The Flags Module code listing in § A.7 on page 210 contains additional documentation.

6.8 Numbers Module

The Numbers Module is used to define number constants that are used in the Cæsar Code Package.

Numbers Module public parameters:

Numbers 0–9	
`zero`	0
`one`	1
`two`	2
`three`	3
`four`	4
`five`	5
`six`	6
`seven`	7
`eight`	8
`nine`	9

Numbers 10–19	
`ten`	10
`eleven`	11
`twelve`	12
`thirteen`	13
`fourteen`	14
`fifteen`	15
`sixteen`	16
`seventeen`	17
`eighteen`	18
`nineteen`	19

Numbers 20-100, by tens	
`twenty`	20
`thirty`	30
`forty`	40
`fifty`	50
`sixty`	60
`seventy`	70
`eighty`	80
`ninety`	90

`hundred`	100

Fractions

`half`	$\frac{1}{2}$
`third`	$\frac{1}{3}$
`fourth`	$\frac{1}{4}$
`fifth`	$\frac{1}{5}$
`sixth`	$\frac{1}{6}$
`seventh`	$\frac{1}{7}$
`eighth`	$\frac{1}{8}$
`ninth`	$\frac{1}{9}$
`tenth`	$\frac{1}{10}$

Forms of pi

`pi`	π
`sqrtpi`	$\sqrt{\pi}$
`invpi`	$\frac{1}{\pi}$
`pisqr`	π^2
`fourthirdspi`	$\frac{4\pi}{3}$
`twopi`	2π
`threepi`	3π
`fourpi`	4π
`halfpi`	$\frac{\pi}{2}$
`thirdpi`	$\frac{\pi}{3}$
`fourthpi`	$\frac{\pi}{4}$
`sqrtfourpi`	$\sqrt{4\pi}$

Decimal multipliers

`deca`	10^1
`hecto`	10^2
`kilo`	10^3
`mega`	10^6
`giga`	10^9
`tera`	10^{12}
`peta`	10^{15}
`exa`	10^{18}
`zetta`	10^{21}
`yotta`	10^{24}
`deci`	10^{-1}
`centi`	10^{-2}
`milli`	10^{-3}
`micro`	10^{-6}
`nano`	10^{-9}
`pico`	10^{-12}
`femto`	10^{-15}
`atto`	10^{-18}

`zepto`	10^{-21}
`yocto`	10^{-24}

The Numbers Module code listing in § A.8 on page 211 contains additional documentation.

Chapter 7

Intrinsics Module

> *The Greek word for leisure was scholê ($\sigma\chi o\lambda\eta$), whence our school. For leisure meant to them opportunity for pursuits of intrinsic worth, such as a man would choose for his own sake – De Burgh*

The Intrinsics Module provides a thin wrapper for the standard F90 intrinsic types (Integer, Real, Character, and Logical), giving them the fundamental procedures necessary to be considered a class (in the Cæsar sense). These fundamental procedures are `Initialize`, `Finalize`, and `Valid_State`. The Cæsar intrinsic types are the pointered versions of standard F90 intrinsic types for multi-dimensional arrays, and non-pointered (static) scalars. The `Valid_State` procedures also have a version for non-pointered arrays, which can be accessed by the name `Valid_State_NP`.

In addition to the fundamental class procedures, the Intrinsics Module provides some functions that are "missing" in F90:

Function	Intrinsic Type
ALL	Logical Scalars
ANY	Logical Scalars
COUNT	Integer/Logical Scalars
MaxVal	Real and Integer Scalars
MinVal	Real and Integer Scalars
SUM	Real and Integer Scalars

and some useful functions involving intrinsics:

Function	Description
.InInterval.	True if argument is in the specified interval.
.InSet.	True if argument is in the specified set.
.NotInInterval.	True if argument is not in the specified interval.
.NotInSet.	True if argument is not in the specified set.
.VeryClose.	True if arguments are within 10 times the local spacing distance between variables.

In addition to the F90 standard intrinsic types, the Status Class is defined here. The Status Class is, in a way, even more basic than the intrinsic types, as it is used in the Initialization and Finalization of the intrinsic types themselves. Several procedures associated with the manipulation of Status objects are included.

The Intrinsics Module code listing in § B on page 213 contains additional documentation.

7.1 Status Class

The Status Class is used to handle status flags in the Cæsar Code Package.

Status public procedures:

Fundamental procedures	
`Initialize`	Initializes a status variable (vector or scalar).
`Finalize`	Finalizes a status variable (vector or scalar).
`Valid_State`	Returns false iff a status variable (vector or scalar) is in an invalid state.
Operations	
`Consolidate`	Consolidates a vector status variable into a scalar status variable (also has an assignment interface).
`Equal`	Returns true when its arguments are equal. Versions of this function for Status-Status and Status-Character comparisons are available.
`Error`	Returns true if the severity level of a scalar status variable is 'Error'.
`Get`	Gets an output character string according to the specified status variable (also has an assignment interface).
`Normal`	Returns true if the severity level of a scalar status variable is 'Normal'.
`Not_Equal`	Returns true when its arguments are not equal. Versions of this function for Status-Status and Status-Character comparisons are available.
`Set`	Sets a scalar status variable according to the specified selector character value (also has an assignment interface).
`Warning`	Returns true if the severity level of a scalar status variable is 'Warning'.

Status public defined types:

`Status_type`	The Status Class variable type.

The Status Class code listing in § B.1 on page 213 contains additional documentation. The Status Class also contains a Unit Test Program which is listed in § B.1.19 on page 231.

7.1.1 Initialize_Status Procedure

The Initialize_Status procedure initializes a status variable.

Calling syntax:

```
call Initialize (S)
```

Output variable:

`S` The status, which has been initialized to a value of "Unset".

The Initialize_Status code listing in § B.1.1 on page 217 contains additional documentation.

7.1.2 Initialize_Status_Vector Procedure

The Initialize_Status_Vector procedure initializes a vector of status variables.

Calling syntax:

```
call Initialize (S)
```

Output variable:

`S` The status vector, which has been initialized to a value of "Unset".

The Initialize_Status_Vector code listing in § B.1.2 on page 217 contains additional documentation.

7.1.3 Finalize_Status Procedure

The Finalize_Status procedure finalizes a status variable.

Calling syntax:

```
call Finalize (S)
```

Input/Output variable:

`S` The status, which has been finalized.

The Finalize_Status code listing in § B.1.3 on page 218 contains additional documentation.

7.1.4 Finalize_Status_Vector Procedure

The Finalize_Status_Vector procedure finalizes a vector of status variables.

Calling syntax:

```
call Finalize (S)
```

Input/Output variable:

`S` The status vector, which has been finalized.

The Finalize_Status_Vector code listing in § B.1.4 on page 219 contains additional documentation.

7.1.5 Valid_State_Status Procedure

The Valid_State_Status procedure returns true iff the status is in a valid state – that is, iff the status passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(S)
```

Input variable:

`S` The status variable to be checked.

Output variable:

`Valid_State` True iff the status is in a valid state.

The Valid_State_Status code listing in § B.1.5 on page 219 contains additional documentation.

7.1.6 Valid_State_Status_Vector Procedure

The Valid_State_Status_Vector procedure returns true iff the status is in a valid state – that is, iff the status passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(S)
```

Input variable:

`S`	The status vector to be checked.

Output variable:

`Valid_State`	True iff the entire status vector is in a valid state.

The Valid_State_Status_Vector code listing in § B.1.6 on page 220 contains additional documentation.

7.1.7 Character_Equal_Status Procedure

The Character_Equal_Status procedure returns a logical value which is true when the status variable is the same state specified by the input character string, used as a selector flag (see Set_Status procedure in section 7.1.13 for a list of valid flags). It has an operator (`==` or `.eq.`) interface.

Calling syntax:

```
Logical = C == SS          or
Logical = Equal(C, SS)
```

Input variable:

`C`	The selector flag string to be compared.
`SS`	The status variable to be compared.

Output variable:

`Equal`	True iff the status variable and the selector flag string match.

The Character_Equal_Status code listing in § B.1.7 on page 221 contains additional documentation.

7.1.8 Character_Not_Equal_Status Procedure

The Character_Not_Equal_Status procedure returns a logical value which is true when the status variable is not equal to the state specified by the input character string, used as a selector flag (see Set_Status procedure in section 7.1.13 for a list of valid flags). It has an operator (`/=` or `.ne.`) interface.

Calling syntax:

```
Logical = C /= SS              or
Logical = Not_Equal(C, SS)
```

Input variable:

`C`	The selector flag string to be compared.
`SS`	The status variable to be compared.

Output variable:

`Not_Equal` True iff the two status variables have different states.

The Character_Not_Equal_Status code listing in § B.1.8 on page 221 contains additional documentation.

7.1.9 Consolidate_Status Procedure

The Consolidate_Status procedure returns a single status variable that is the result of consolidating an input status vector. It has an assignment interface so that it may be called by assigning a vector Status_type variable to a scalar Status_type variable.

The input status vector (Multiple_S) is combined into the output status variable (Consolidated_S) by looping over each input status variable. The following table shows the value of Consolidated_S after it has been combined with a single value from the vector Multiple_S, based on the previous value of Consolidated_S:

Consolidated_S	Multiple_S(i)					
(previous)	Unset	Success	ME	MW	Error	Warning
Unset	Unset	Success	ME	MW	Error	Warning
Success	Success	Success	ME	MW	Error	Warning
ME	ME	ME	ME	ME	ME	ME
MW	MW	MW	ME	MW	ME	MW
Error	Error	Error	ME	ME	ME†	ME
Warning	Warning	Warning	ME	MW	ME	MW†

where ME stands for 'Multiple Error', MW stands for 'Multiple Warning' and † signifies that ME or MW is set only if the two errors or warnings are different from each other.

Calling syntax:

```
Consolidated_S = Multiple_S
call Consolidate (Consolidated_S, Multiple_S)
```

or

Input variable:

`Multiple_S` A vector of status variables to be consolidated.

Output variable:

`Consolidated_S` The status that is a result of consolidating the input status vector.

The Consolidate_Status code listing in § B.1.9 on page 222 contains additional documentation.

7.1.10 Error_Status Procedure

The Error_Status procedure returns true iff the status is an error status – that is, iff the status severity level is 'error'.

Calling syntax:

```
Logical = Error(S)
```

Input variable:

`S` The status variable to be examined.

Output variable:

`Error` True iff the status is an error status.

The Error_Status code listing in § B.1.10 on page 225 contains additional documentation.

7.1.11 Get_Status_Output Procedure

The Get_Status_Output procedure returns the output string from a status variable. It has an assignment interface so that it may be called by assigning a Status_type variable to a character variable.

Calling syntax:

```
Status_String = S              or
call Get (Status_String, S)
```

Input variable:

`S` The status to be examined.

Output variable:

`Status_String` Output string for this status variable.

The Get_Status_Output code listing in § B.1.11 on page 226 contains additional documentation.

7.1.12 Normal_Status Procedure

The Normal_Status procedure returns true iff the status is a normal status – that is, iff the status severity level is 'normal'.

Calling syntax:

```
Logical = Normal(S)
```

Input variable:

`S` The status variable to be examined.

Output variable:

`Normal` True iff the status is a normal status.

The Normal_Status code listing in § B.1.12 on page 226 contains additional documentation.

7.1.13 Set_Status Procedure

The Set_Status procedure assigns a value to a status variable. It has an assignment interface so that it may be called by assigning a character variable to a Status_type variable.

Calling syntax:

```
S = Selector_Flag              or
call Set (S, Selector_Flag)
```

Input variable:

`Selector_Flag`	String to select status flag. Possible values are: 'Unset', 'Success', 'Multiple Error', 'Multiple Warning', 'Memory Warning', 'Memory Error', 'File Error', 'CGNS Error', 'CGNS No Node', 'CGNS Bad Path'.

Output variable:

`S`	The status, which is now set.

The Set_Status code listing in § B.1.13 on page 227 contains additional documentation.

7.1.14 Status_Equal_Character Procedure

The Status_Equal_Character procedure returns a logical value which is true when the status variable is the same state specified by the input character string, used as a selector flag (see Set_Status procedure in section 7.1.13 for a list of valid flags). It has an operator (== or `.eq.`) interface.

Calling syntax:

```
Logical = S == C          or
Logical = Equal(S, C)
```

Input variable:

`C`	The selector flag string to be compared.
`S`	The status variable to be compared.

Output variable:

`Equal`	True iff the status variable and the selector flag string match.

The Status_Equal_Character code listing in § B.1.14 on page 228 contains additional documentation.

7.1.15 Status_Equal_Status Procedure

The Status_Equal_Status procedure returns a logical value which is true when the two status variable inputs are equal (i.e. have the same state). It has an operator (== or `.eq.`) interface.

Calling syntax:

```
Logical = S1 == S2          or
Logical = Equal(S1, S2)
```

Input variable:

`S1, S2`	The status variables to be compared.

Output variable:

`Equal`	True iff the two status variables have the same state.

The Status_Equal_Status code listing in § B.1.15 on page 228 contains additional documentation.

7.1.16 Status_Not_Equal_Character Procedure

The Status_Not_Equal_Character procedure returns a logical value which is true when the status variable is not equal to the state specified by the input character string, used as a selector flag (see Set_Status procedure in section 7.1.13 for a list of valid flags). It has an operator (`/=` or `.ne.`) interface.

Calling syntax:

```
Logical = S /= C            or
Logical = Not_Equal(S, C)
```

Input variable:

`C`	The selector flag string to be compared.
`S`	The status variable to be compared.

Output variable:

`Not_Equal`	True iff the two status variables have different states.

The Status_Not_Equal_Character code listing in § B.1.16 on page 229 contains additional documentation.

7.1.17 Status_Not_Equal_Status Procedure

The Status_Not_Equal_Status procedure returns a logical value which is true when the two status variable inputs are not equal (i.e. have different states). It has an operator (`/=` or `.ne.`) interface.

Calling syntax:

```
Logical = S1 /= S2            or
Logical = Not_Equal(S1, S2)
```

Input variable:

`S1, S2`	The status variables to be compared.

Output variable:

`Not_Equal`	True iff the two status variables have different states.

The Status_Not_Equal_Status code listing in § B.1.17 on page 230 contains additional documentation.

7.1.18 Warning_Status Procedure

The Warning_Status procedure returns true iff the status is a warning status – that is, iff the status severity level is ‘warning’.

Calling syntax:

```
Logical = Warning(S)
```

Input variable:

`S` The status variable to be examined.

Output variable:

`Warning` True iff the status is a warning status.

The Warning_Status code listing in § B.1.18 on page 230 contains additional documentation.

7.2 Real Class

The Real Class is used to describe a real scalar or array in the CÆSAR Code Package. The Real Class does not use a user-defined type – it is the class for the F90 intrinsic type "real".

Real public procedures:

Fundamental procedures

`Initialize`	Initializes a real scalar or array variable.
`Finalize`	Finalizes a real scalar or array variable.
`Valid_State`	Returns false iff a real scalar or array variable is in an invalid state.

Operations

`MaxVal`	Extends the F90 intrinsic procedure MaxVal to scalar arguments.
`MinVal`	Extends the F90 intrinsic procedure MinVal to scalar arguments.
`SUM`	Extends the F90 intrinsic procedure SUM to scalar arguments.
`VeryClose`	Returns true if input values are almost equal.

The Real Class code listing in § B.2 on page 233 contains additional documentation. The Real Class also contains a Unit Test Program which is listed in § B.2.8 on page 244.

7.2.1 Initialize_Real Procedure

The Initialize_Real procedure allocates and initializes a real scalar or array variable.

Calling syntax:

```
call Initialize (R, [dim1, ..., dimn,] status)
```

Input variables:

`R`	A real scalar variable or a pointer to an unallocated real array variable.
`dim1, ..., dimn`	Extents of the dimensions for the array R. Only as many dimensions as are needed should be entered.

Output variables:

`R`	The R variable has been allocated (if it is an array) and initialized to `initialize_real_flag`.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variable:

`allocate_status`	Allocation Status.

The Initialize_Real code listing in § B.2.1 on page 234 contains additional documentation.

7.2.2 Finalize_Real Procedure

The Finalize_Real procedure deallocates and finalizes a real scalar or array variable.

Calling syntax:

```
call Finalize (R, status)
```

Input variable:

`R`	A real scalar variable or a pointer to an allocated real array variable.

Output variables:

`R`	The R variable has been deallocated and nullified, if it is an array, or set to a flag value (`finalize_real_flag`), if it is a scalar.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variable:

`deallocate_status`	Deallocation Status.

The Finalize_Real code listing in § B.2.2 on page 236 contains additional documentation.

7.2.3 Valid_State_Real Procedure

The Valid_State_Real procedure returns true iff the real scalar or array variable is in a valid state – that is, iff the variable passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(R)       or
Logical = Valid_State_NP(R)
```

Input variable:

`R`	A real scalar variable or a pointer to an allocated real array variable. A non-pointered variable may be used with the Valid_State_NP version of the call.

Output variable:

`Valid_State`	True iff the real scalar or array variable is in a valid state.

The Valid_State_Real code listing in § B.2.3 on page 238 contains additional documentation.

7.2.4 MaxVal_Real_Scalar Procedure

The MaxVal_Real_Scalar procedure provides a function equivalent to the MaxVal() intrinsic function for real scalars. It returns a value equal to the input value.

Calling syntax:

```
Real = MaxVal(R)
```

Input variable:

`R` Input real scalar variable.

Output variable:

`MaxVal` MaxVal is equal to R.

The MaxVal_Real_Scalar code listing in § B.2.4 on page 242 contains additional documentation.

7.2.5 MinVal_Real_Scalar Procedure

The MinVal_Real_Scalar procedure provides a function equivalent to the MinVal() intrinsic function for real scalars. It returns a value equal to the input value.

Calling syntax:

```
Real = MinVal(R)
```

Input variable:

`R` Input real scalar variable.

Output variable:

`MinVal` MinVal is equal to R.

The MinVal_Real_Scalar code listing in § B.2.5 on page 242 contains additional documentation.

7.2.6 SUM_Real_Scalar Procedure

The SUM_Real_Scalar procedure provides a function equivalent to the SUM() intrinsic function for real scalars. It returns a value equal to the input value.

Calling syntax:

```
Real = SUM(R)
```

Input variable:

`R` Input real scalar variable.

Output variable:

`SUM` SUM is equal to R.

The SUM_Real_Scalar code listing in § B.2.6 on page 243 contains additional documentation.

7.2.7 VeryClose_Real Procedure

The VeryClose procedure provides a way to check whether or not two variables are "Very Close" to each other. The definition of "Very CLose" that is used is "within 10 times the internumeral spacing distance in the vicinity of the numbers". This test should be used instead of equality tests for reals. For array-valued X and Y, all the values must satisfy the condition for the result to be true.

Note that I would rather call this routine "Very_Close", but the F90 standard does not allow underscores in defined operator names (i.e. ".Very_Close." is not allowed).

Calling syntax:

```
Logical = X .VeryClose.  Y    ,
Logical = VeryClose(X, Y)     or
```

Input variables:

`X, Y`	Input real, scalar or array variables to test.

Output variable:

`VeryClose`	Logical which is true if X and Y are within 10 times the spacing distance between variables at (X+Y)/2.

The VeryClose_Real code listing in § B.2.7 on page 243 contains additional documentation.

7.3 Integer Class

The Integer Class is used to describe an integer scalar or array in the Cæsar Code Package. The Integer Class does not use a user-defined type – it is the class for the F90 intrinsic type "integer".

Integer public procedures:

Fundamental procedures

`Initialize`	Initializes an integer scalar or array variable.
`Finalize`	Finalizes an integer scalar or array variable.
`Valid_State`	Returns false iff an integer scalar or array variable is in an invalid state.

Operations

`MaxVal`	Extends the F90 intrinsic procedure MaxVal to scalar arguments.
`MinVal`	Extends the F90 intrinsic procedure MinVal to scalar arguments.
`SUM`	Extends the F90 intrinsic procedure SUM to scalar arguments.

The Integer Class code listing in § B.3 on page 246 contains additional documentation. The Integer Class also contains a Unit Test Program which is listed in § B.3.7 on page 254.

7.3.1 Initialize_Integer Procedure

The Initialize_Integer procedure allocates and initializes an integer scalar or array variable.

Calling syntax:

```
call Initialize (I, [dim1, ..., dimn,] status)
```

Input variables:

`I`	An integer scalar variable or a pointer to an unallocated integer array variable.
`dim1, ..., dimn`	Extents of the dimensions for the array I. Only as many dimensions as are needed should be entered.

Output variables:

`I`	The I variable has been allocated (if it is an array) and initialized to `initialize_integer_flag`.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variable:

`allocate_status`	Allocation Status.

The Initialize_Integer code listing in § B.3.1 on page 248 contains additional documentation.

7.3.2 Finalize_Integer Procedure

The Finalize_Integer procedure deallocates and finalizes an integer scalar or array variable.

Calling syntax:

```
call Finalize (I, status)
```

Input variable:

`I`	An integer scalar variable or a pointer to an allocated integer array variable.

Output variable:

`I`	The I variable has been deallocated and nullified, if it is an array, or set to a flag value (`finalize_integer_flag`), if it is a scalar.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variable:

`deallocate_status`	Deallocation Status.

The Finalize_Integer code listing in § B.3.2 on page 249 contains additional documentation.

7.3.3 Valid_State_Integer Procedure

The Valid_State_Integer procedure returns true iff the integer scalar or array variable is in a valid state – that is, iff the variable passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(I)       or
Logical = Valid_State_NP(I)
```

Input variable:

`I` An integer scalar variable or a pointer to an allocated integer array variable. A non-pointered variable may be used with the Valid_State_NP version of the call.

Output variable:

`Valid_State` True iff the integer scalar or array variable is in a valid state.

The Valid_State_Integer code listing in § B.3.3 on page 251 contains additional documentation.

7.3.4 MaxVal_Integer_Scalar Procedure

The MaxVal_Integer_Scalar procedure provides a function equivalent to the MaxVal() intrinsic function for integer scalars. It returns a value equal to the input value.

Calling syntax:

```
Integer = MaxVal(I)
```

Input variable:

`I` Input integer scalar variable.

Output variable:

`MaxVal` MaxVal is equal to I.

The MaxVal_Integer_Scalar code listing in § B.3.4 on page 252 contains additional documentation.

7.3.5 MinVal_Integer_Scalar Procedure

The MinVal_Integer_Scalar procedure provides a function equivalent to the MinVal() intrinsic function for integer scalars. It returns a value equal to the input value.

Calling syntax:

```
Integer = MinVal(I)
```

Input variable:

`I` Input integer scalar variable.

Output variable:

`MinVal` MinVal is equal to I.

The MinVal_Integer_Scalar code listing in § B.3.5 on page 253 contains additional documentation.

7.3.6 SUM_Integer_Scalar Procedure

The SUM_Integer_Scalar procedure provides a function equivalent to the SUM() intrinsic function for integer scalars. It returns a value equal to the input value.

Calling syntax:

```
Integer = SUM(I)
```

Input variable:

`I` Input integer scalar variable.

Output variable:

`SUM` SUM is equal to I.

The SUM_Integer_Scalar code listing in § B.3.6 on page 253 contains additional documentation.

7.4 Logical Class

The Logical Class is used to describe a logical scalar or array in the CÆSAR Code Package. The Logical Class does not use a user-defined type – it is the class for the F90 intrinsic type "logical".

Logical public procedures:

Fundamental procedures	
`Initialize`	Initializes a logical scalar or array variable.
`Finalize`	Finalizes a logical scalar or array variable.
`Valid_State`	Returns false iff a logical scalar or array variable is in an invalid state.
Operations	
`ALL`	Extends the F90 intrinsic procedure ALL to scalar arguments.
`ANY`	Extends the F90 intrinsic procedure ANY to scalar arguments.
`COUNT`	Extends the F90 intrinsic procedure COUNT to scalar arguments.
`InInterval`	Returns true iff the argument is in the specified interval.
`InSet`	Returns true iff the argument is in the specified set.
`NotInInterval`	Returns true iff the argument is not in the specified interval.
`NotInSet`	Returns true iff the argument is not in the specified set.

The Logical Class code listing in § B.4 on page 256 contains additional documentation. The Logical Class also contains a Unit Test Program which is listed in § B.4.11 on page 268.

7.4.1 Initialize_Logical Procedure

The Initialize_Logical procedure allocates and initializes a logical scalar or array variable.

Calling syntax:

```
call Initialize (L, [dim1, ..., dimn,] status)
```

Input variables:

`L`	A logical scalar variable or a pointer to an unallocated logical array variable.
`dim1, ..., dimn`	Extents of the dimensions for the array L. Only as many dimensions as are needed should be entered.

Output variables:

`L`	The L variable has been allocated (if it is an array) and initialized to `initialize_logical_flag`.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variable:

`allocate_status`	Allocation Status.

The Initialize_Logical code listing in § B.4.1 on page 258 contains additional documentation.

7.4.2 Finalize_Logical Procedure

The Finalize_Logical procedure deallocates and finalizes a logical scalar or array variable.

Calling syntax:

```
call Finalize (L, status)
```

Input variable:

`L`	A logical scalar variable or a pointer to an allocated logical array variable.

Output variable:

`L`	The L variable has been deallocated and nullified, if it is an array, or set to a flag value (`finalize_logical_flag`), if it is a scalar.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variable:

`deallocate_status`	Deallocation Status.

The Finalize_Logical code listing in § B.4.2 on page 260 contains additional documentation.

7.4.3 Valid_State_Logical Procedure

The Valid_State_Logical procedure returns true iff the logical scalar or array variable is in a valid state – that is, iff the variable passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(L)        or
Logical = Valid_State_NP(L)
```

Input variable:

`L` A logical scalar variable or a pointer to an allocated logical array variable. A non-pointered variable may be used with the Valid_State_NP version of the call.

Output variable:

`Valid_State` True iff the logical scalar or array variable is in a valid state.

The Valid_State_Logical code listing in § B.4.3 on page 261 contains additional documentation.

7.4.4 ALL_Scalar Procedure

The ALL_Scalar procedure provides a function equivalent to the ALL() intrinsic function for logical scalars. It returns true iff its input is true.

Calling syntax:

```
Logical = ALL(L)
```

Input variable:

`L` Input logical scalar variable.

Output variable:

`ALL` ALL is true iff L is true.

The ALL_Scalar code listing in § B.4.4 on page 262 contains additional documentation.

7.4.5 ANY_Scalar Procedure

The ANY_Scalar procedure provides a function equivalent to the ANY() intrinsic function for logical scalars. It returns true iff its input is true.

Calling syntax:

```
Logical = ANY(L)
```

Input variable:

`L` Input logical scalar variable.

Output variable:

`ANY`	ANY is true iff L is true.

The ANY_Scalar code listing in § B.4.5 on page 263 contains additional documentation.

7.4.6 COUNT_Scalar Procedure

The COUNT_Scalar procedure provides a function equivalent to the COUNT() intrinsic function for logical scalars. It returns the number of trues in the input, so it is 1 if the input scalar is true and 0 if the input scalar is false.

Calling syntax:

```
Integer = COUNT(L)
```

Input variable:

`L`	Input logical scalar variable.

Output variable:

`COUNT_Scalar`	Set to the number of trues in the input (either 0 or 1).

The COUNT_Scalar code listing in § B.4.6 on page 263 contains additional documentation.

7.4.7 InInterval Procedure

The InInterval procedure provides a way to check whether or not a value lies within a specified interval. The interval is considered to be a closed interval that includes the end-points. For an array-valued X, all the values must satisfy the condition for the result to be true.

Note that I would rather call this routine "In_Interval", but the F90 standard does not allow underscores in defined operator names (i.e. ".In_Interval." is not allowed).

Calling syntax:

```
Logical = X .InInterval.  (/Int1, Int2/)    ,
Logical = X .InInterval.  Interval          or
Logical = InInterval(X, Interval)
```

Input variables:

`X`	Input integer or real, scalar or array variable.
`Interval`	A vector of length 2 that specifies the extents of the interval to be checked.
`(/Int1, Int2/)`	A means of expressing an interval without declaring a vector.

Output variable:

`InInterval`	Logical which is true if X is in the closed interval, which includes the end-points.

The InInterval code listing in § B.4.7 on page 264 contains additional documentation.

7.4.8 InSet Procedure

The InSet procedure provides a way to check whether or not a value lies within a specified set.

Note that I would rather call this routine "In_Set", but the F90 standard does not allow underscores in defined operator names (i.e. ".In_Set." is not allowed).

Calling syntax:

```
Logical = X .InSet.  (/Elem1, Elem2, .../)    ,
Logical = X .InSet.  Set                      or
Logical = InSet(X, Set)
```

Input variables:

`X`	Input integer, real or character scalar variable.
`Set`	A vector containing the elements of the set to be checked.
`(/Elem1, Elem2, .../)`	A means of expressing a set without declaring a vector.

Output variable:

`InSet`	Logical which is true if X is an element of the set.

The InSet code listing in § B.4.8 on page 265 contains additional documentation.

7.4.9 NotInInterval Procedure

The NotInInterval procedure provides a way to check whether or not a value lies outside of a specified interval. The interval is considered to be a closed interval that includes the end-points. For an array-valued X, all the values must satisfy the condition for the result to be true.

Note that I would rather call this routine "Not_In_Interval", but the F90 standard does not allow underscores in defined operator names (i.e. ".Not_In_Interval." is not allowed).

Calling syntax:

```
Logical = X .NotInInterval.  (/Int1, Int2/)    ,
Logical = X .NotInInterval.  Interval          or
Logical = NotInInterval(X, Interval)
```

Input variables:

`X`	Input integer or real, scalar or array variable.
`Interval`	A vector of length 2 that specifies the extents of the interval to be checked.
`(/Int1, Int2/)`	A means of expressing an interval without declaring a vector.

Output variable:

`NotInInterval`	Logical which is true if X is in the closed interval, which includes the end-points.

The NotInInterval code listing in § B.4.9 on page 266 contains additional documentation.

7.4.10 NotInSet Procedure

The NotInSet procedure provides a way to check whether or not a value lies within a specified set.

Note that I would rather call this routine "Not_In_Set", but the F90 standard does not allow underscores in defined operator names (i.e. ".Not_In_Set." is not allowed).

Calling syntax:

```
Logical = X .NotInSet.  (/Elem1, Elem2, .../)   ,
Logical = X .NotInSet.  Set                     or
Logical = NotInSet(X, Set)
```

Input variables:

`X`	Input integer, real or character scalar variable.
`Set`	A vector containing the elements of the set to be checked.
`(/Elem1, Elem2, .../)`	A means of expressing a set without declaring a vector.

Output variable:

`NotInSet`	Logical which is true if X is not an element of the set.

The NotInSet code listing in § B.4.10 on page 267 contains additional documentation.

7.5 Character Class

The Character Class is used to describe a character scalar or array in the CÆSAR Code Package. The Character Class does not use a user-defined type – it is the class for the F90 intrinsic type "character".

Character public procedures:

Fundamental procedures	
`Initialize`	Initializes a character scalar or array variable.
`Finalize`	Finalizes a character scalar or array variable.
`Valid_State`	Returns false iff a character scalar or array variable is in an invalid state.

The Character Class code listing in § B.5 on page 269 contains additional documentation. The Character Class also contains a Unit Test Program which is listed in § B.5.4 on page 275.

7.5.1 Initialize_Character Procedure

The Initialize_Character procedure allocates and initializes a character scalar or array variable.

Calling syntax:

```
call Initialize (C, [dim1, ..., dimn,] status)
```

Input variables:

`C`	A character scalar variable or a pointer to an unallocated character array variable.
`dim1, ..., dimn`	Extents of the dimensions for the array C. Only as many dimensions as are needed should be entered.

Output variables:

`C`	The C variable has been allocated (if it is an array) and initialized to `initialize_-character_flag`.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variable:

`allocate_status`	Allocation Status.

The Initialize_Character code listing in § B.5.1 on page 270 contains additional documentation.

7.5.2 Finalize_Character Procedure

The Finalize_Character procedure deallocates and finalizes a character scalar or array variable.

Calling syntax:

```
call Finalize (C, status)
```

Input variable:

`C`	A character scalar variable or a pointer to an allocated character array variable.

Output variable:

`C`	The C variable has been deallocated and nullified, if it is an array, or set to a flag value (`finalize_character_flag`), if it is a scalar.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`deallocate_status`	Deallocation Status.

The Finalize_Character code listing in § B.5.2 on page 272 contains additional documentation.

7.5.3 Valid_State_Character Procedure

The Valid_State_Character procedure returns true iff the character scalar or array variable is in a valid state – that is, iff the variable passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(C)       or
Logical = Valid_State_NP(C)
```

Input variable:

`C` A character scalar variable or a pointer to an allocated character array variable. A non-pointered variable may be used with the Valid_State_NP version of the call.

Output variable:

`Valid_State` True iff the character scalar or array variable is in a valid state.

The Valid_State_Character code listing in § B.5.3 on page 274 contains additional documentation.

Chapter 8

Utilities Module

Robin: *Where'd you get a live fish, Batman?*
Batman: *The true crimefighter always carries everything he needs in his utility belt, Robin.*

There are some procedures in CÆSAR that do not belong to a particular class, yet are still generally useful. It does not make sense to develop a class around these procedures unless one wishes to be pedantic about object-oriented programming, so they are collected in the Utilities Module.

They are grouped by function, with three of the groups, Shell_Utils, File_Utils and Text_Utils, based on duplicating the Gnu shell functions in Fortran95 code. As of April 4, 2003, these three Gnu shell function groupings were combined into a package called Core_Utils, but they will continue to be grouped separately here.

The Utilities Module code listing in § C on page 277 contains additional documentation.

8.1 F2003_Utils Module

The F2003_Utils Module provides intrinsic procedures that conform to the Fortran 2003 standard[1]. The procedures provided are similar, but may not behave exactly the same as the F2003 versions. These functions should be replaced by F2003 calls as compilers become available.

F2003_Utils public procedures:

`Command_Argument_Count`	Returns the number of arguments on the command line.
`Get_Command_Argument`	Returns a particular argument number from the command line.

The F2003_Utils Module code listing in § C.1 on page 277 contains additional documentation. The F2003_-Utils Module also contains a Unit Test Program which is listed in § C.1.3 on page 280.

8.1.1 Command_Argument_Count_F2003 Procedure

The Command_Argument_Count_F2003 returns the number of command-line arguments used for this invocation of the executable. It is modeled after the Fortran 2003 function so that it can be replaced by that function eventually.

Calling syntax:

[1] `http://j3-fortran.org/`

```
Integer = COMMAND_ARGUMENT_COUNT()
```

(This is capitalized because it is an intrinsic in Fortran 2003.)

Output variable:

`Command_Argument_Count`	Returns the number of command arguments.

The Command_Argument_Count_F2003 code listing in § C.1.1 on page 278 contains additional documentation.

8.1.2 Get_Command_Argument_F2003 Procedure

The Get_Command_Argument_F2003 procedure returns a particular command-line argument used for this invocation of the executable. It is modeled after the Fortran 2003 function so that it can be replaced by that function eventually.

Calling syntax:

```
call GET_COMMAND_ARGUMENT(Number, Argument)
```

(This is capitalized because it is an intrinsic in Fortran 2003.)

Input variables:

`Number`	The number of the argument to be queried.

Output variable:

`Argument`	The character value of the argument.

The Get_Command_Argument_F2003 code listing in § C.1.2 on page 279 contains additional documentation.

8.2 Shell_Utils Module

The Shell_Utils Module provides utility routines that mimic those found in the Gnu shell-utils package for use inside Fortran routines. The routines provided are similar, but may not behave exactly the same as the Gnu versions.

Candidates for inclusion in this module (i.e., the utilities in the Gnu version) are: basename, chroot, date, dirname, echo, env, expr, factor, false, groups, hostname, id, logname, nice, nohup, pathchk, printenv, printf, pwd, seq, sleep, stty, su, tee, test, true, tty, uname, users, who, whoami, and yes.

Shell_Utils public procedures:

`Basename`	Removes the path prefix and optionally the suffix from a given pathname.
`Dirname`	Strips off the filename or last level and returns only the directory name from a given pathname.

The Shell_Utils Module code listing in § C.2 on page 280 contains additional documentation. The Shell_Utils Module also contains a Unit Test Program which is listed in § C.2.3 on page 283.

8.2.1 Basename_Shell_Utils Procedure

The Basename_Shell_Utils procedure strips off the path prefix and optionally the suffix from a given pathname. Note that the suffix behavior is slightly different from the standard basename function.

Calling syntax:

```
Character = Basename(Filename, Suffix)
```

Input variables:

`Filename`	The Filename to be modified.
`Suffix_Strip`	Toggle for removing suffix. Default is true. [Optional]

Output variable:

`Basename`	The Filename with any leadin pathname and (optionally) any suffix stripped off.

Internal variables:

`basename_left`	Leftmost character of stripped name.
`basename_right`	Rightmost character of stripped name.

The Basename_Shell_Utils code listing in § C.2.1 on page 281 contains additional documentation.

8.2.2 Dirname_Shell_Utils Procedure

The Dirname_Shell_Utils procedure strips off the filename and returns only the directory name from a given pathname.

Calling syntax:

```
Character = Dirname(Filename)
```

Input variables:

`Filename`	The Filename to be modified.

Output variable:

`Dirname`	The directory part of the input Filename. Note that '.' is returned if there is no directory part.

Internal variables:

`dirname_right`	Rightmost character of directory name.

The Dirname_Shell_Utils code listing in § C.2.2 on page 283 contains additional documentation.

8.3 Text_Utils Module

Change changing places
Root yourself to the ground
Capitalize on this good fortune
One word can bring you round
Changes
– Yes, 90125, "Changes"

The Text_Utils Module provides utility routines that mimic those found in the Gnu text-utils package for use inside Fortran routines. The routines provided are similar, but may not behave exactly the same as the Gnu versions. The Text_Utils Module also provides various other functions to manipulate text.

Candidates for inclusion in this module (i.e., the utilities in the Gnu version) are: cat, cksum, comm, csplit, cut, expand, fmt, fold, head, join, md5sum, nl, od, paste, ptx, pr, sort, split, sum, tac, tail, tr, tsort, unexpand, uniq, and wc.

Text_Utils public procedures:

`Capitalize`	Converts a string to lowercase with uppercase at the beginning of words.
`Lowercase`	Converts a string to all lowercase.
`Uppercase`	Converts a string to all uppercase.

The Text_Utils Module code listing in § C.3 on page 285 contains additional documentation. The Text_Utils Module also contains a Unit Test Program which is listed in § C.3.4 on page 289.

8.3.1 Capitalize_Text_Utils Procedure

The Capitalize_Text_Utils procedure returns a capitalized version of the input string. That is, the string is all in lowercase except for the first letter of every word, which is uppercase. Words are defined as contiguous strings of letters, and all non-letter characters define word boundaries.

Calling syntax:

```
Character = Capitalize(String)
```

Input variables:

`String`	The String to be capitalized.

Output variable:

`Capitalize`	The capitalized version of the input String.

The Capitalize_Text_Utils code listing in § C.3.1 on page 286 contains additional documentation.

8.3.2 Lowercase_Text_Utils Procedure

The Lowercase_Text_Utils procedure returns a lowercased version of the input string.

Calling syntax:

```
Character = Lowercase(String)
```

Input variables:

`String`	The String to be lowercased.

Output variable:

`Lowercase`	The lowercase version of the input String.

The Lowercase_Text_Utils code listing in § C.3.2 on page 287 contains additional documentation.

8.3.3 Uppercase_Text_Utils Procedure

The Uppercase_Text_Utils procedure returns an uppercased version of the input string.

Calling syntax:

```
Character = Uppercase(String)
```

Input variables:

`String`	The String to be uppercased.

Output variable:

`Uppercase`	The uppercase version of the input String.

The Uppercase_Text_Utils code listing in § C.3.3 on page 288 contains additional documentation.

Chapter 9

Data_Structures Module

Get your data structures correct first, and the rest of the program will write itself. – David Jones

The secret of life is data structures. – Elizabeth Schwarzin

Data structures are important. Programs with less than optimal data structures experience problems reaching their full potential, and programs with poor data structures can fail completely. Parallel data structures are even more important, and even more complex. Ideally, this complexity should be hidden from the casual user.

The Data_Structures Module implements the fundamental parallel data structures in the CÆSAR Code Package, and addresses the following issues:

- Input/Output on a single processor
- Data distribution over all the PEs
- Indirect addressing – gather/scatters
- Encapsulation – hiding the tricky operations
- Data access and storage
- User-specified base structures
- Memory/communication time trade-offs

The rest of this section explains how this is accomplished.

Communication and Trace Classes

First, all of the low-level communication calls in CÆSAR are wrapped by the Communication Class (described in § 9.2 on page 73). These calls currently use the PGSLib Package (described in § 2.2.2 on page 8), but they could be rewritten to use UPS or even bare calls from the MPI Package (described in § 2.2.1 on page 8). The Communication Class also includes a serial version. Another data structure auxiliary type, which contains the information associated with a gather-scatter operation, is the Trace Class (described in § 9.1 on page 69).

Strategy

Every data structure in the Data Structures Module can be thought of as a multi-dimensional array, with one dimension spread across the processors. The dimension that is spread across the processors is referred

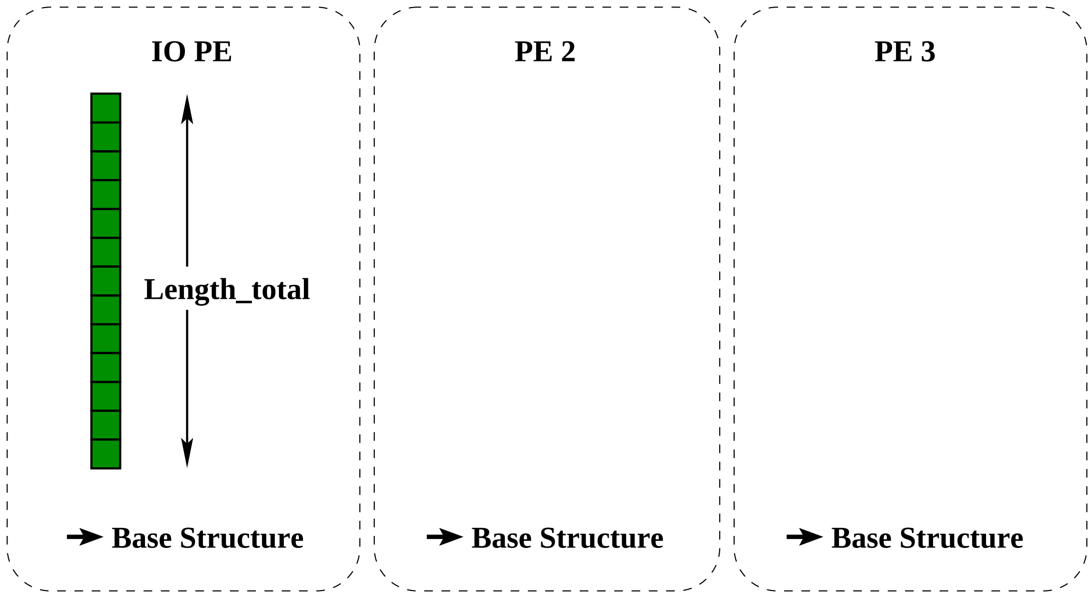

Figure 9.1: A Schematic Diagram of the Assembled Vector data structure. The data has been *assembled* on the IO PE.

to as the distributed (or parallel) dimension (or axis). [1] The remaining serial axes are contained on a given processor, and are treated like standard Fortran 95 arrays. In the rest of this discussion the serial axes will be ignored and arrays will be described as vectors, with the understanding that a "vector" may actually have several serial axes in addition to its single parallel or distributed axis. Note also that the PEs are assumed to contain contiguous pieces of the vector, with the first section being on PE=1, the second section being on PE=2, etc.

Base_Structure Class

Information about the basic distribution of an axis across the PEs is contained in the Base_Structure Class (described in § 9.3 on page 79). This information includes items such as the total length of the vector, the length of the vector on this PE, the starting and ending indices for this PE, and the locus (or name) for the axis. For example, a user might specify a Base Structure for the nodes in a mesh (locus = 'Nodes'), another one for the cells, and another one for the faces. Or equivalently, Base Structures could be defined for equations or variables. The actual locus is not specified by the Data_Structures Module, so the user may easily define new ones as needed.

Assembled_Vector Class

The simplest CÆSAR data structure, the Assembled_Vector Class (described in § 9.5 on page 89), is not parallel at all, but exists only on a single processor (see Figure 9.1). These vectors can be thought of as parallel data structures that have been "assembled" on a single PE (the IO processor). The primary use for an Assembled Vector is input and output, although there may certainly be times when a given parallel data structure is too large to be assembled on the IO PE. An Assembled Vector contains a pointer to a Base Structure which determines its structure.

Distributed_Vector Class

The basic parallel data structure is defined by the Distributed_Vector Class (described in § 9.6 on page 94). A Distributed Vector is a true parallel data structure, with data spread across the processors according to its Base Structure, a different amount on each PE (see Figure 9.2). A Distributed Vector contains a pointer to a Base Structure which determines its structure. The conversions between Assembled Vectors

[1] This is a similar idea to the distribution specified by a CMF array with a layout containing a single :NEWS (parallel) axis (*CM Fortran Reference Manual*, 1989).

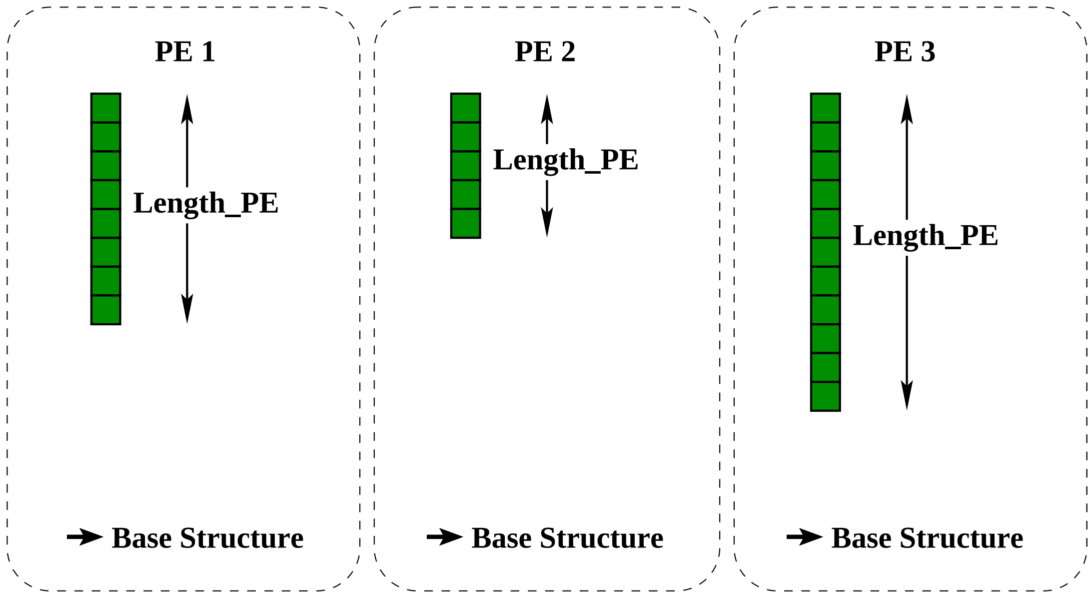

Figure 9.2: A Schematic Diagram of the Distributed Vector data structure. The data has been *distributed* to all of the PEs, a different amount on each one.

and Distributed Vectors (based on the same Base Structure) can be accomplished with an equals sign (via operator overloading). The operation changing a Distributed Vector to an Assembled Vector is known as *assemble*; the opposite operation is called *distribute*.

Assembled and Distributed Vectors, and the Base Structures that they are based upon, satisfy the input/output and parallel distribution needs of CÆSAR. There is also a need for the *association* of two different distributions. This is used to define, for example, the nodes that correspond to each cell in a mesh. The association between two data distributions can also be thought of as *indirection*, which is very useful for unstructured mesh operations (or parallel distributed mesh operations).

Data_Index Class

Association between two Base Structures in CÆSAR is accomplished via the Data_Index Class (described in § 9.4 on page 84). This association is considered to be a many-of-one relationship – "many" entries in the first Base Structure correspond to "one" entry in the second Base Structure. For example, many nodes in a node Base Structure correspond to each cell in a cell Base Structure. A Data Index therefore contains a Many (Base) Structure, a One (Base) Structure, and an index array to relate the two. It also stores some information about the communication pattern necessary to gather the "many" entries for each "one" entry on the one axis.

Note that "many" need not be more than unity – for instance there could be one boundary face for each regular face. In this case, the index array would be one-dimensional. If "many" is a constant number (for instance, there are always six faces on each cell), then the index array is a two-dimensional array (in this example, six by the number of cells on each PE). If "many" is a variable number (for instance, a polyhedral mesh where the number of faces per cell may vary), then a ragged-right index array is used. Ragged-right index arrays are not yet implemented, but one- and two-dimensional arrays are.

Overlapped_Vector Class

The Overlapped_Vector Class (described in § 9.7 on page 100) defines a data structure that represents an intermediate step in performing a gather and collect operation. The data in an Overlapped Vector has been gathered from a Distributed Vector with a Many Structure according to a specified One Structure, but has not been collected into place. An Overlapped Vector contains a Many (Base) Structure, a One (Base) Structure, a Many-of-One (Data) Index, and a Distributed Vector with a Many Structure (see Figure 9.3). It

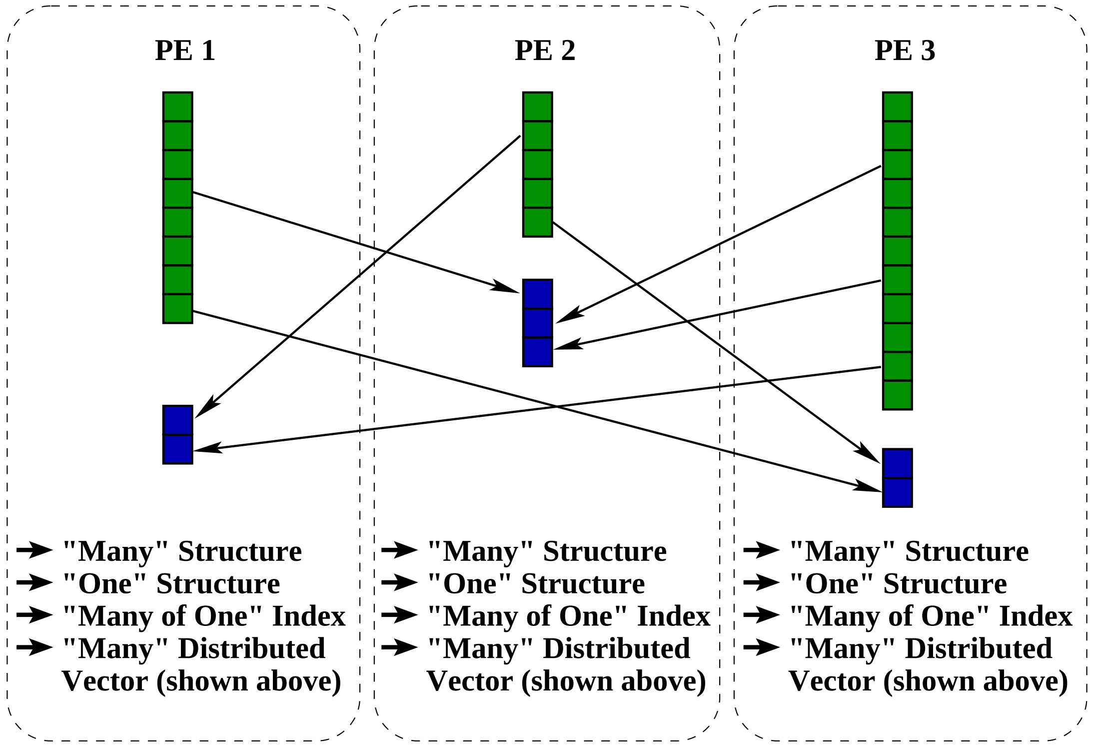

Figure 9.3: A Schematic Diagram of the Overlapped Vector data structure. The green blocks represent the Distributed Vector that the Overlapped Vector includes. The blue blocks represent off-processor data necessary for a gather operation that has been stored locally – the *overlapped* part of the vector.

also contains the entries from the Distributed Vector that would be needed for a gather and collect operation that are not local, and so some data is represented multiple times – but no more than once on a given PE. This *overlapped* data can represent the data from the boundary of a given processor that does not reside locally in the Distributed Vector. For example, an Overlapped Vector can contain the coordinates for all the nodes that correspond to cells on a given processor, even though some of the nodes on the boundary actually reside on other processors. Since it does not store the nodes twice on a given processor (even if more than one cell on that processor contains a specific node), the Overlapped Vector data structure uses a smaller amount of memory (compared to a Collected Array, which is described in the next subsection). Since it does not require interprocessor communication to generate the collected form of the data, an Overlapped Vector can save on run time (compared to gathering and collecting a Distributed Vector).

Collected_Array Class

The fully-evaluated "many-of-one" relationship between two distributions is described by the Collected_-Array Class (described in § 9.8 on page 107). In contrast to the other Cæsar data structures, the Collected Array is an *array* rather than a vector (see Figure 9.4). It has *two* axes to represent the single parallel axis that has been discussed so far (in addition to the many possible serial axes whose discussion is being suppressed here). The vertical axis in Figure 9.4 is distributed across the processors according to the One Structure. The horizontal axis of the Collected Array has been formed by *collecting* all of the "many" items that correspond to each "one" (note that a Collected Array for a two-dimensional Many-of-One Index is shown).

A Collected Array is the final result of a gather and collect operation on a Distributed Vector according to a Many-of-One Index, or the result of simply *collecting* an Overlapped Vector (with no interprocessor communication). The data in a Collected Array is in a very useable form, so that access requires little run-time (as compared to collecting an Overlapped Vector or gathering and collecting a Distributed Vector). The data in a Collected Array takes up much more memory than an Overlapped Vector, since "many" items

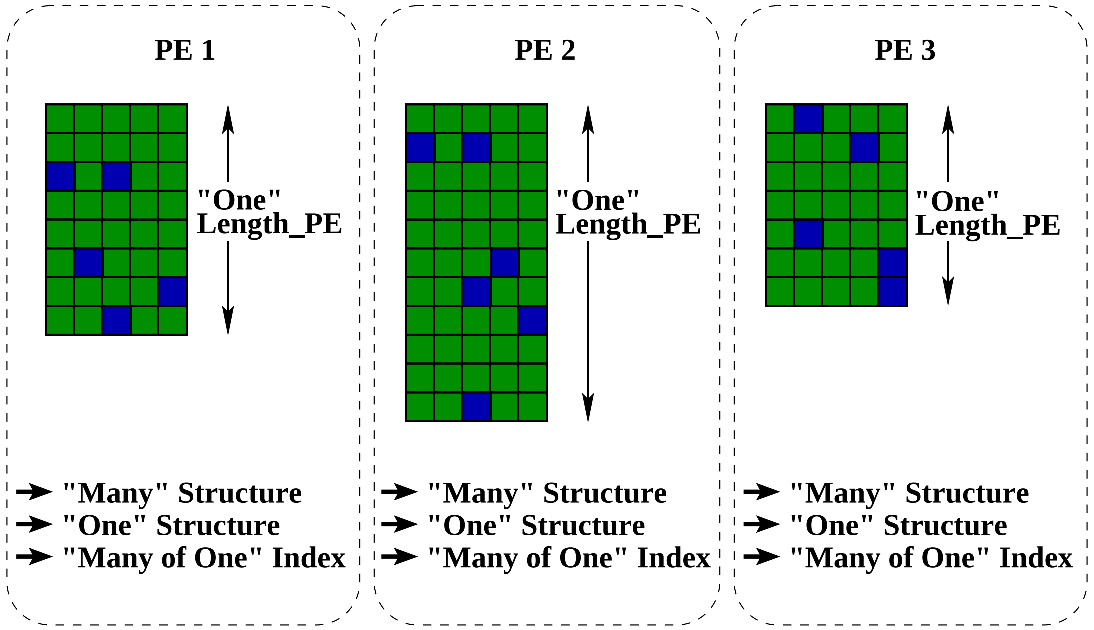

Figure 9.4: A Schematic Diagram of the Collected Array data structure. One way to form a Collected Array is to *collect* all of the "many" entries that correspond to each entry on the "one" axis. The green blocks represent data that was already on-processor when the data was distributed on the many axis in a Distributed Vector. The blue blocks represent off-processor data that was gathered and collected onto the one axis, which is the same as the blue overlapped blocks in an Overlapped Vector.

that belong to more than a single "one" item (for example, nodes belonging to more than one cell) are stored multiply, even if they exist on the same processor. Sometimes this behavior is not only useful from a run-time perspective, but is required – for instance, a physical property evaluated on each face of a cell may require a different value for the same face in different cells due to different materials in the cells. The "many" axis in a Collected Array can be *combined* using a specified combination operator to form a Distributed Vector with a One Structure.

Bare Naked Vectors and Arrays

In contrast to the communication, the calculations using data from the data structures are done using standard Fortran 95 arrays, pointered to allow dynamic memory allocation. This allows the compiler to make any optimizations possible with the Fortran 95 intrinsic array syntax. For the purpose of this discussion, to contrast these standard Fortran arrays with the other data structures described here, they are referred to as Bare Naked Vectors and Bare Naked Arrays, meaning that there is no encumbering derived type associated with them.

Summary

The CÆSAR data structures are very useful in coordinating the necessary communication for a parallel program. They provide a great deal of control over data layout, memory usage and cpu time, and enable CÆSAR to optimize in one direction or another depending on available resources (see Table 9.1). All of the complicated operations of converting from one data structure to another are hidden, and in fact can in most cases be accomplished with a simple equals sign, via operator overloading.

An Example

As an example, take the problem of reading in node coordinates, calculating cell center values, and writing them out. The steps to be taken are diagrammed in Figures 9.5 and 9.6. First, set-up information for the nodes is read in, in the form of the total number of nodes and the number on each PE, and a Base Structure for the nodes (referred to as the Node Structure) is initialized. A similar set up is done for the cells. Next,

Table 9.1: Relative Data Structure Memory and CPU Requirements. Assuming that information is stored in a Distributed Vector and needs to be accessed in a gathered and collected Bare Naked Array form, this table shows the memory / cpu time trade-offs for various CÆSAR data structures.

Data Structure	Memory Usage	CPU Usage
Distributed Vector	Lowest	High, with communication
Overlapped Vector	Low (same as DV + off-PE entries)	Medium, no communication
Collected Array	High	Low, no communication

an index array that tells which nodes are associated with which cells is read, and a Data Index called the Nodes of Cells Index is initialized using the index array and the Node and Cell Structures.

Now the necessary data structures can be easily initialized from the Node Structure, the Cell Structure, and the Nodes of Cells Index. These are, using acronyms for the data structures (i.e. AV: Assembled Vector, DV: Distributed Vector, OV: Overlapped Vector, CA: Collected Array, BNV: Bare Naked Vector, BNA: Bare Naked Array):

Coordinates_Nodes_BNV (IO PE only),
Coordinates_Nodes_AV,
Coordinates_Nodes_DV,
Coordinates_Nodes_of_Cells_OV,
Coordinates_Nodes_of_Cells_CA,
Centers_Cells_DV,
Centers_Cells_AV, and
Centers_Cells_BNV (IO PE only).

All of these data structures have a single serial axis in addition to the parallel axis, and it is dimensioned to the number of spatial dimensions in the problem (3 for 3-D).

The node coordinates are now read into a Bare Naked Vector called Coordinates_Nodes_BNV which is defined on the IO PE only. To store the coordinates in the Assembled Vector, a simple equals sign is used:

```
Coordinates_Nodes_AV = Coordinates_Nodes_BNV  .
```

To distribute the values across the processors, use another equals sign:

```
Coordinates_Nodes_DV = Coordinates_Nodes_AV  .
```

To gather the distributed node values to their respective cells, so that all nodes are on the same processor with their cells, in an Overlapped Vector, again use an equals sign:

```
Coordinates_Nodes_of_Cells_OV = Coordinates_Nodes_DV  .
```

To collect the node coordinates for each cell into an array, still an equals sign:

```
Coordinates_Nodes_of_Cells_CA = Coordinates_Nodes_of_Cells_OV  .
```

To calculate the cell center coordinates, combine the node coordinates for each cell using an "Average" operator. In this case, a subroutine call must be used to specify the combination operator:[2]

```
call Combine_with_Average (Centers_Cells_DV, Coordinates_Nodes_of_Cells_CA)  .
```

To assemble the cell center coordinates on the IO PE, use an equals sign:

```
Centers_Cells_AV = Centers_Cells_DV  .
```

To access the cell center coordinates on the IO PE, use a final equals sign:

```
Centers_Cells_BNV = Centers_Cells_AV  .
```

And then the cell center coordinates may be written to a file, completing our example.

[2]An equals sign could have been used if the combination operator had been a "SUM".

If the DEBUG_LEVEL is set high enough, copious error checking will be done to make sure that the left and right sides of the equals signs above are compatible.

Note that the solution given above is not unique. For example, to conserve memory, the Collected Array need not be formed, and a direct step around it may be taken:

```
call Collect_and_Average (Centers_Cells_DV, Coordinates_Nodes_of_Cells_OV)  .
```

Or, if the memory is available, the intermediate Overlapped Vector is superfluous and can be skipped:

```
Coordinates_Nodes_of_Cells_CA = Coordinates_Nodes_DV  .
```

Forming an Overlapped Vector does not use much more memory than the Distributed Vector it is based on, and saves much communication time.

The Data_Structures Module code listing in § D on page 291 contains additional documentation.

9.1 Trace Class

The Trace Class is used to describe a communication trace in the CÆSAR Code Package. A communication trace consists of the set-up information for a particular gather-scatter call.

Trace public procedures:

Fundamental procedures	
`Initialize`	Initializes a Trace object.
`Finalize`	Finalizes a Trace object.
`Valid_State`	Returns false iff a Trace object is in an invalid state.
Operations	
`Initialized`	Returns true iff a Trace object has been initialized.

Trace public defined type:

Trace type	
`Dimensionality`	The number of dimensions that the index has.
`Index1, Index2`	The index values, which may be modified by the communications package.
`Initialized`	Initialization status.
`Trace`	The trace for the communication associated with the index.

The Trace Class code listing in § D.1 on page 292 contains additional documentation.

9.1.1 Initialize_Trace Procedure

The Initialize_Trace procedure allocates and initializes a Trace object.

Calling syntax:

```
call Initialize (Trace, Index, Length_PE, status)
```

Input variables:

`Trace`	The Trace object to be initialized.
`Index`	The indirect index values for the specified communication pattern.
`Length_PE`	Length of the destination vector on this PE. This corresponds to the Many axis of a Many-to-One relationship.

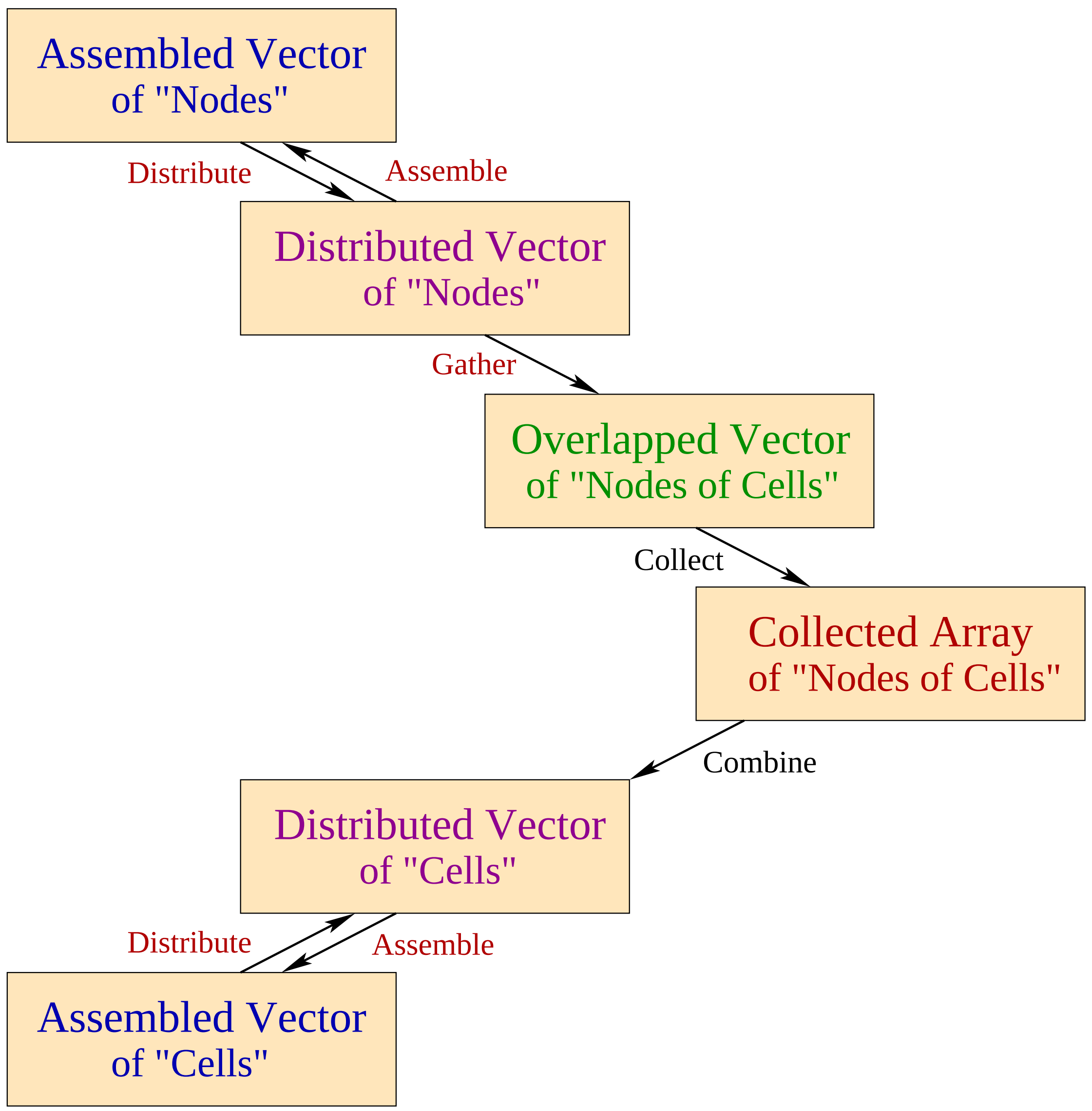

Figure 9.5: A flow chart showing the hierarchical relationships between CÆSAR Data Structures. Operations in red require global communication.

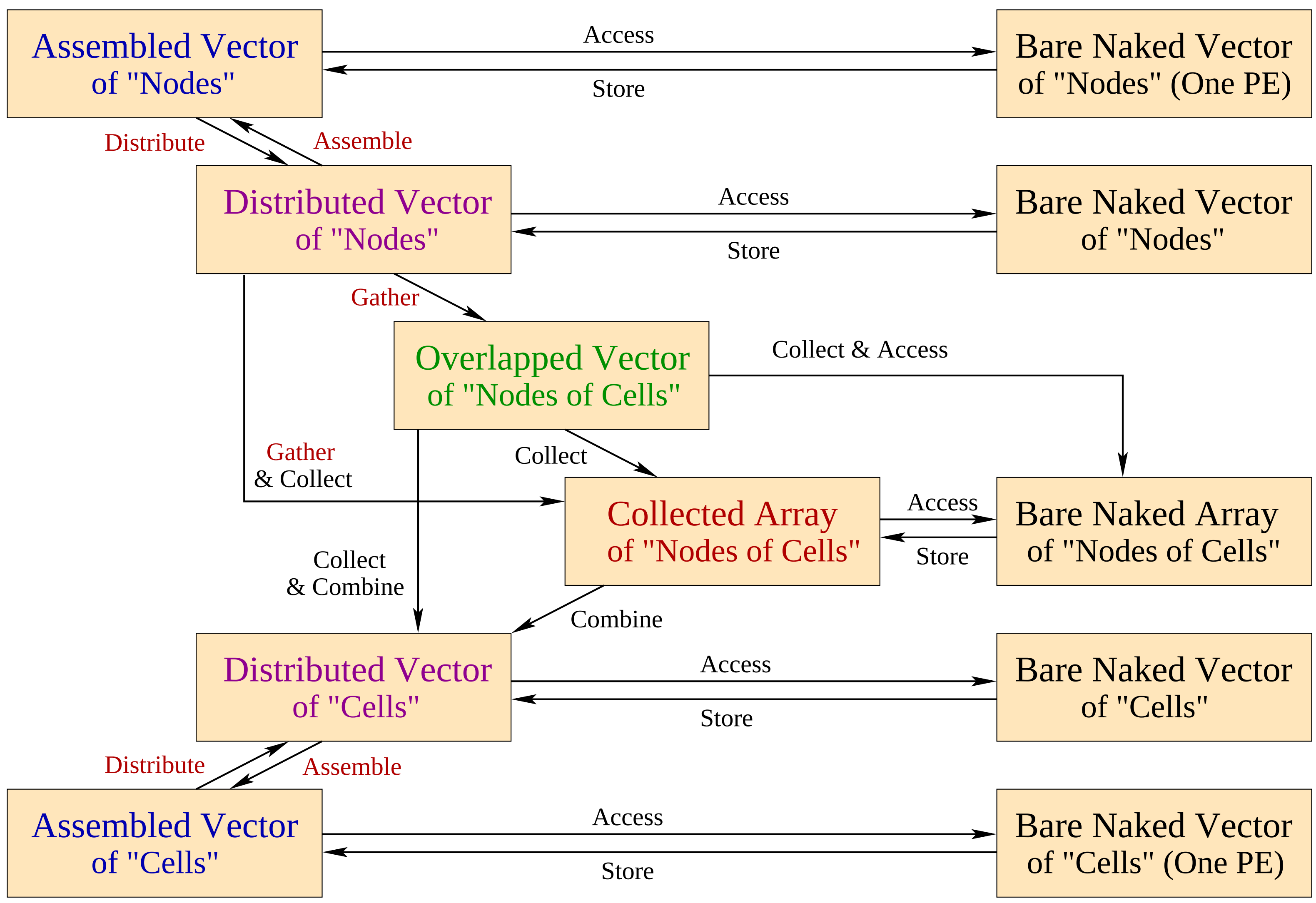

Figure 9.6: A flow chart showing operations that have been implemented for CÆSAR Data Structures. Operations in red require global communication.

Output variables:

`Trace`	The Trace object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.

The Initialize_Trace code listing in § D.1.1 on page 293 contains additional documentation.

9.1.2 Finalize_Trace Procedure

The Finalize_Trace procedure deallocates and finalizes a Trace object.

Calling syntax:

```
call Finalize (Trace, status)
```

Input variables:

`Trace`	The Trace object to be finalized.

Output variables:

`Trace`	The Trace object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Trace code listing in § D.1.2 on page 295 contains additional documentation.

9.1.3 Valid_State_Trace Procedure

The Valid_State_Trace procedure returns true iff the Trace is in a valid state – that is, iff the Trace passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Trace)
```

Input variables:

`Trace`	The Trace to be checked.

Output variable:

`Valid_State`	True iff the Trace is in a valid state.

The Valid_State_Trace code listing in § D.1.3 on page 297 contains additional documentation.

9.1.4 Initialized_Trace Procedure

The Initialized_Trace procedure returns true iff the Trace object has been initialized.

Calling syntax:

```
Logical = Initialized(Trace)
```

Input variable:

`Trace`	The Trace object to be examined.

Output variable:

`Initialized`	True iff the Trace object has been initialized.

The Initialized_Trace code listing in § D.1.4 on page 298 contains additional documentation.

9.2 Communication Class

The Communication Class is used to wrap the communication calls to other packages from the CÆSAR Code Package. It also contains serial versions of the routines so that CÆSAR may be run without any external packages on a serial platform.

Communication public procedures:

Fundamental procedures	
`Initialize`	Initializes communication.
`Finalize`	Finalizes communication.
`Valid_State`	Returns false iff the communication is in an invalid state.
Operations	
`Abort`	Stops execution on all processors.
`Assemble`	Takes a variable that is distributed across all the processors and pulls it together (assembles it) on the IO PE. This is the opposite of the `Distribute` procedure.
`Broadcast`	Sets a variable on all the processors to a value on the IO PE.
`Distribute`	Takes a variable on the IO PE and divides it up (distributes it) among all the processors. This is the opposite of the `Assemble` procedure.
`Gather`	Transforms one distributed variable into another distributed variable, according to an indirection index. No collisions are possible, since this call is effectively pulling values out of a distributed variable, and there is a different location for each pulled value. This is the opposite of the `Scatter` procedure.
`Global Reductions`	Global operations that require communication with all the processors, such as SUM, ALL, etc.
`Output`	Outputs information about the communication set-up.
`Output_Test`	Outputs the result of a test.
`Parallel_Write`	Outputs data that is distributed across the processors.

`Scatter_OP`	Transforms one distributed variable into another distributed variable, according to an indirection index. Collisions are possible, since this call is effectively putting values into a distributed variable, and more than one value can go into the same location. Therefore, a combination operator, OP, must be specified. Allowed values for OP are: `AND`, `MAX`, `MIN`, `OR`, or `SUM`. This is the opposite of the `Gather` procedure.

Communication public defined types:

`Communication_type`	The Communication Class variable type.

Communication public variables:

`delta_PE_IO_PE`	Kronecker delta (PE, IO_PE).
`IO_PE`	The PE number which is allowed to do I/O.
`NPEs`	Total number of PEs (1 for serial runs).
`Parallel`	True for parallel runs, false for serial runs.
`Parallel_Library`	The name of the parallel communication library.
`Serial`	True for serial runs, false for parallel runs.
`Scope`	Global scope for PGSLib – not really utilized yet.
`this_is_IO_PE`	True on any PE which is allowed to perform I/O operations.
`this_is_not_IO_PE`	True on any PE which is not allowed to perform I/O operations.
`this_PE`	The PE number for this processor (in the range [1, `NPEs`]).

The Communication Class code listing in § D.2 on page 298 contains additional documentation. The Communication Class also contains a Unit Test Program which is listed in § D.2.14 on page 323.

9.2.1 Initialize_Communication Procedure

The Initialize_Communication procedure sets up and initializes the communication scheme for the CÆSAR Code Package. Once it has been called, the global communication variables are defined and available via use association. This routine should be called whether or not a parallel run is being done.

Calling syntax:

```
call Initialize (Communication)
```

Input/Output variable:

`Communication`	The Communication object to be initialized.

The Initialize_Communication code listing in § D.2.1 on page 302 contains additional documentation.

9.2.2 Finalize_Communication Procedure

The Finalize_Communication procedure finalizes the communication scheme for the CÆSAR Code Package.

Calling syntax:

```
call Finalize (Communication, Output_Toggle)
```

Input variable:

`Output_Toggle`	Toggles final output for this communication set-up. [Optional]

Input/Output variable:

`Communication`	The Communication object to be finalized.

The Finalize_Communication code listing in § D.2.2 on page 303 contains additional documentation.

9.2.3 Valid_State_Communication Procedure

The Valid_State_Communication procedure returns true iff the communication is in a valid state – that is, iff the communication passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Communication)
```

Input variable:

`Communication`	The Communication object to be checked.

Output variable:

`Valid_State`	True iff the communication is in a valid state.

The Valid_State_Communication code listing in § D.2.3 on page 305 contains additional documentation.

9.2.4 Abort Procedure

The Abort Procedure stops program execution. In parallel mode, Abort should stop execution on all processors.

Calling syntax:

```
call Abort
```

The Abort code listing in § D.2.4 on page 306 contains additional documentation.

9.2.5 Assemble Procedure

The Assemble Procedure takes an input vector or scalar that is distributed across all the processors and pulls it together (assembles it) on the IO processor. This is the opposite of the Distribute procedure.

Calling syntax:

```
call Assemble (Output, Input)
```

Input variable:

`Input`	A real, integer, logical or character vector or scalar that is distributed across all the processors and which is to be assembled on the IO processor.

Output variable:

`Output`	The assembled version of the `Input` variable, existing only on the IO processor.

The Assemble code listing in § D.2.5 on page 306 contains additional documentation.

9.2.6 Broadcast Procedure

The Broadcast Procedure takes an input value from the IO processor and sets the variable on all of the processors equal to that value.

Calling syntax:

```
call Broadcast (Variable)
```

Input variable:

`Variable` A real, integer, logical or character vector that is defined on the IO processor and which is to be broadcast over all of the processors.

Output variable:

`Variable` Every processor now has the same value for variable, equal to the input value on the IO processor.

The Broadcast code listing in § D.2.6 on page 307 contains additional documentation.

9.2.7 Distribute Procedure

The Distribute Procedure takes an input vector from the IO processor and divides it up (distributes it) across all of the processors. This is the opposite of the Assemble procedure.

Calling syntax:

```
call Distribute (Output, Input, Lengths)
```

Input variable:

`Input` A real, integer or logical vector that is defined on the IO processor and which is to be distributed over all of the processors.

`Lengths` An integer vector of dimension `NPEs` containing the number of elements of the input vector which are to be distributed to each PE. Lengths is only defined on the IO PE. In the special case of one element to each processor, this vector must not be included.

Output variable:

`Output` The distributed version of the `Input` vector, with a piece on every processor.

The Distribute code listing in § D.2.7 on page 308 contains additional documentation.

9.2.8 Gather Procedure

The Gather Procedure takes an input vector that is distributed across all the processors and gathers it into another distributed vector or array according to an indirect index vector or array. No collisions are possible, since this call is effectively pulling values out of a distributed variable, and there is a different location for each pulled value. This is the opposite of the Scatter procedure.

Calling syntax:

```
call Gather (Output, Input, Index, Trace)
```

Input variables:

`Index`	An optional integer vector or array of indirect references to positions in the `Input` vector. This must be included on the first call to this procedure with a given data structure, but may be omitted on subsequent calls if the `Trace` variable is present. [Optional]
`Input`	A real, integer or logical vector that is distributed across all the processors.
`Trace`	An optional structure that stores the setup from a previous Gather/Scatter call using the same `Index` variable and `Input` vector length. If `Trace` is present and uninitialized, it is set by this procedure. If Trace is present, it is used regardless of whether Index is present. [Optional]

Output variables:

`Output`	The gathered version of the `Input` vector, distributed across the processors.
`Trace`	If present, `Trace` is set to the setup information for this Gather/Scatter. [Optional]

The Gather code listing in § D.2.8 on page 310 contains additional documentation.

9.2.9 Global Reduction Functions

The global reduction functions execute various global reductions on their input. Each global reduction function corresponds to a Fortran 90 intrinsic function with the same name, minus the "Global_" prefix. These functions require global all-to-all communication and are relatively computationally expensive. If the code is run in serial mode, then the global reduction functions are equivalent to the serial intrinsic functions.

Calling syntax:

```
Output = Global_ALL (Input)                    ,
Output = Global_ANY (Input)                    ,
Output = Global_MaxVal (Input)                 ,
Output = Global_MinVal (Input)                 ,
Output = Global_Sum (Input)                    or
Output = Global_Dot_Product (Input1, Input2)
```

Input variables:

`Input`	A real or integer (in the case of Sum, MinVal and MaxVal) or a logical (in the case of ALL and ANY) scalar, vector or 2-D array that is defined on each processor and which is to be reduced over all of the processors.
`Input1, Input2`	The two real, integer or logical vectors that are defined on each processor and that are to be reduced via a dot product over all of the processors.

Output variable:

`Output`	The result of the global reduction.

The Global Reduction code listing in § D.2.9 on page 313 contains additional documentation.

9.2.10 Output_Communication Procedure

The Output_Communication procedure writes out information about the communication set-up to the specified unit.

Calling syntax:

```
call Output (Communication, Unit)
```

Input variables:

`Communication`	The Communication object to be output.
`Unit`	The logical unit for output, which defaults to 6. [Optional]

The Output_Communication code listing in § D.2.10 on page 315 contains additional documentation.

9.2.11 Output_Test Procedure

The Output_Test procedure writes out the result of a test to the specified unit.

Calling syntax:

```
call Output_Test (Test_Name, Success, Unit)
```

Input variables:

`Test_Name`	The name of the test that has been conducted.
`Success`	The result status of the test.
`Unit`	The logical unit for output, which defaults to 6. [Optional]

The Output_Test code listing in § D.2.11 on page 316 contains additional documentation.

9.2.12 Parallel_Write Procedure

The Parallel_Write Procedure takes an input character scalar or vector that is distributed across all the processors and writes it to the specified unit number. If a specific PE number is specified, only the information for that PE is written; otherwise, the information for all the PEs is written in order.

Calling syntax:

```
call Parallel_Write (String, Unit, PE)
```

Input variables:

`String`	The character string (scalar or vector) to be written out, defined differently on each processor.
`PE`	The processor number containing the data to be output. If not present, data from all of the processors will be output. [Optional]
`Unit`	The unit number for output. If not present, unit 6 (`stdout`) will be used. [Optional]

The Parallel_Write code listing in § D.2.12 on page 317 contains additional documentation.

9.2.13 Scatter Procedure

The Scatter Procedure takes an input vector (bare naked vector) that is distributed across all the processors and scatters it into another distributed vector or array (bare naked vector or array) according to an indirect index vector or array. Collisions are possible, since this call is effectively putting values into a distributed variable, and more than one value can go into the same location. Therefore, a combination operator, OP, must be specified. This is the opposite of the Gather procedure.

Calling syntax:

```
call Scatter_AND (Output, Input, Index, Trace)   ,
call Scatter_MAX (Output, Input, Index, Trace)   ,
call Scatter_MIN (Output, Input, Index, Trace)   ,
call Scatter_OR (Output, Input, Index, Trace)    or
call Scatter_SUM (Output, Input, Index, Trace)
```

Input variables:

`Index`	An optional integer vector or array of indirect references to positions in the `Output` vector. This must be included on the first call to this procedure with a given data structure, but may be omitted on subsequent calls if the `Trace` variable is present. [Optional]
`Input`	A real, integer or logical vector that is distributed across all the processors.
`Trace`	An optional structure that stores the setup from a previous Gather/Scatter call using the same `Index` variable and `Output` vector length. If `Trace` is present and uninitialized, it is set by this procedure. If Trace is present, it is used regardless of whether Index is present. [Optional]

Output variables:

`Output`	The scattered vector version of the `Input` vector, distributed across the processors.
`Trace`	If present, `Trace` is set to the setup information for this Gather/Scatter. [Optional]

The Scatter code listing in § D.2.13 on page 320 contains additional documentation.

9.3 Base_Structure Class

The Base_Structure Class is used to describe a base structure in the Cæsar Code Package. A Base_Structure is part of the overall data structure strategy in Cæsar, which is made up of the following classes: Base_Structure, Data_Index, Assembled_Vector, Distributed_Vector, Overlapped_Vector, and Collected_Array. A description of the overall data structure strategy can be found in the Data_Structures Module (described in chapter 9 on page 63).

Base_Structure public procedures:

Fundamental procedures	
`Initialize`	Initializes a Base_Structure object.
`Finalize`	Finalizes a Base_Structure object.
`Valid_State`	Returns false iff a Base_Structure object is in an invalid state.
`Initialized`	Returns true iff a Base_Structure object has been initialized.
Operations	
`First_PE`	Returns the first global index number on this PE.
`Generate_Even_Distribution`	Returns an even distribution of items in a vector.

`Generate_Multiple`	Generates a Base_Structure object which is a multiple of another Base_Structure object.
`Last_PE`	Returns the last global index number on this PE.
`Length_PE`	Returns the length of the distributed axis on this PE.
`Length_Total`	Returns the total the distributed axis of the entire vector (including all PEs).
`Length_Vector`	Returns a vector containing the length of the distributed axis for each PE.
`Locus`	Returns the locus of the Base_Structure object.
`Output`	Writes out the Base_Structure object.
`Range_PE`	Returns the range of global index numbers on this PE.

Base_Structure public variable:

`name_length`	Length of the character strings for names.

Base_Structure public defined type:

Base_Structure type	
`First_PE`	First global index number for this PE.
`Initialized`	Initialization status.
`Last_PE`	Last global index number for this PE.
`Length_Total`	Total length of the distributed axis of the entire vector (including all PEs).
`Length_PE`	Length of the distributed axis on this PE.
`Length_Vector`	A vector containing the length of the distributed axis for each PE.
`Locus`	The location or variable name which is distributed over the processors (Cells, Nodes, Faces, Equations, Variables, etc.).
`Range_PE`	The range of global index numbers for this PE.

The Base_Structure Class code listing in § D.3 on page 324 contains additional documentation. The Base_-Structure Class also contains a Unit Test Program which is listed in § D.3.9 on page 338.

9.3.1 Initialize_Base_Structure Procedure

The Initialize_Base_Structure procedure allocates and initializes a Base_Structure object.

Calling syntax:

```
call Initialize (Structure, Length_Vector, Locus, status)
```

Input variables:

`Structure`	The Base_Structure object to be initialized.
`Length_Vector`	A vector containing the length of the distributed axis for each PE.
`Locus`	The location or variable name which is distributed over the processors (Cells, Nodes, Faces, Equations, Variables, etc.). [Optional]

Output variables:

`Structure`	The Base_Structure object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.

The Initialize_Base_Structure code listing in § D.3.1 on page 327 contains additional documentation.

9.3.2 Finalize_Base_Structure Procedure

The Finalize_Base_Structure procedure deallocates and finalizes a Base_Structure object.

Calling syntax:

```
call Finalize (Structure, status)
```

Input variables:

`Structure`	The Base_Structure object to be finalized.

Output variables:

`Structure`	The Base_Structure object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Base_Structure code listing in § D.3.2 on page 328 contains additional documentation.

9.3.3 Valid_State_Base_Structure Procedure

The Valid_State_Base_Structure procedure returns true iff the Base_Structure is in a valid state – that is, iff the Base_Structure passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Structure)
```

Input variables:

`Structure`	The Base_Structure to be checked.

Output variable:

`Valid_State`	True iff the Base_Structure is in a valid state.

The Valid_State_Base_Structure code listing in § D.3.3 on page 329 contains additional documentation.

9.3.4 Initialized_Base_Structure Procedure

The Initialized_Base_Structure procedure returns true iff the Base_Structure object has been initialized.

Calling syntax:

```
Logical = Initialized(Structure)
```

Input variable:

`Structure`	The Base_Structure object to be examined.

Output variable:

`Initialized`	True iff the Base_Structure object has been initialized.

The Initialized_Base_Structure code listing in § D.3.4 on page 330 contains additional documentation.

9.3.5 Generate_Even_Distribution Procedure

The Generate_Even_Distribution procedure sets a short vector (on the order of NPEs) to the most even distribution of items possible, on every PE. It can be used to generate an even distribution for the Length_-Vector to initialize a Base_Structure object.

Calling syntax:

```
call Generate_Even_Distribution (Vector, NItems)
```

Input variable:

`NItems`	The number of items to be distributed.

Output variable:

`Vector`	The vector with an even distribution of items.

The Generate_Even_Distribution code listing in § D.3.5 on page 331 contains additional documentation.

9.3.6 Generate_Multiple_Base_Structure Procedure

The Generate_Multiple_Base_Structure procedure generates a structure which is a multiple of an existing Base Structure – that is, it initializes a new Base Structure that is N times as large as the old Base Structure on every PE.

Calling syntax:

```
call Generate_Multiple (Structure_Multiple, N, Structure_source, status)
```

Input variables:

`Structure_source`	The Base_Structure to be duplicated.
`N`	The number of times larger the new structure is to be.
`Locus`	The location or variable name which is distributed over the processors (Cells, Nodes, Faces, Equations, Variables, etc.). [Optional]

Output variables:

`Structure_Multiple`	The Base_Structure object is now a copy of the Structure_source Mathematic Vector.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

The Generate_Multiple_Base_Structure code listing in § D.3.6 on page 332 contains additional documentation.

9.3.7 Get Value Base_Structure Functions

The Get_Value_Structure functions return values from a Base_Structure object.

Calling syntax:

```
Output = First_PE (Structure)         ,
Output = Last_PE (Structure)          ,
Output = Length_PE (Structure)        ,
Output = Length_Total (Structure)     ,
Output = Length_Vector (Structure)    ,
Output = Locus (Structure)            or
Output = Range_PE (Structure)
```

Input variables:

`Structure`	The Base_Structure object to be examined.

Output variable:

`Output`	For Locus, returns a character variable containing the locus assigned to the object upon initialization. For Range_PE, returns a dimension(2) integer containing the range of values on this PE. For Length_Vector, returns a dimension(NPEs) integer containing the number of values on all PEs. For all other functions, returns an integer variable with the named value for the Base_Structure object.

The Get Value Base_Structure code listing in § D.3.7 on page 334 contains additional documentation.

9.3.8 Output_Base_Structure Procedure

The Output_Base_Structure procedure writes out a Base Structure to the specified unit.

Calling syntax:

```
call Output (Structure, Unit, Type, Indent)
```

Input variables:

`Structure`	The Base_Structure object to be queried.
`Unit`	The logical unit for output, which defaults to 6. [Optional]
`Type`	The structure type (e.g. Many, One). [Optional]
`Indent`	Number of indentation characters. [Optional]

The Output_Base_Structure code listing in § D.3.8 on page 336 contains additional documentation.

9.4 Data_Index Class

The Data_Index Class is used to describe a data index in the CÆSAR Code Package. A Data_Index is part of the overall data structure strategy in CÆSAR, which is made up of the following classes: Base_Structure, Data_Index, Assembled_Vector, Distributed_Vector, Overlapped_Vector, and Collected_Array. A description of the overall data structure strategy can be found in the Data_Structures Module (described in chapter 9 on page 63).

The form of a Data_Index object is given by:

```
Array ( One_Axis [, Many_Axis] )
```

or

```
Array ( One_Axis [, Many_Axis_ragged_right] )
--> not implemented
```

where One_Axis refers to the axis which is spread across the processors.

Data_Index public procedures:

Fundamental procedures	
`Initialize`	Initializes a Data_Index object.
`Finalize`	Finalizes a Data_Index object.
`Valid_State`	Returns false iff a Data_Index object is in an invalid state.
`Initialized`	Returns true iff a Data_Index object has been initialized.
Operations	
`Get_Values`	Gets the values from a Data_Index object and returns them in a bare naked vector (also has an assignment interface).
`Initialize_Shell_Partition`	Sets up the Base_Structure and Data_Index objects for a poor partitioning, used for testing.
`Output`	Writes out the Data_Index object.

Data_Index public defined type:

Data_Index type	
`Dimensionality`	The number of dimensions that the index has. "Ragged_Right" indices are signified by a Dimensionality of -1, and are equivalent to a Dimensionality of 2 where the number of columns per row varies. (Ragged_Right is not yet implemented.)
`Index1, Index2`	The index values, which are modified: 1. to reflect off-PE locations with a negative number corresponding to the location in Off_PE_Index of the original value; and 2. to have a local numbering instead of a global numbering.
`Initialized`	Initialization status.
`Many_Structure`	Basic data structure corresponding to the columns in the index array. The index array can be thought of as a "Many of One" relationship (e.g. Many Faces of Each Cell, or Faces_of_Cells), with each row of the array signifying all the "Many" items that correspond to that "One" row.
`NOff_PE`	The number of Off-PE values.
`Off_PE_Index`	The values of the index which are not local to this PE.
`Off_PE_Trace`	The trace for the communication associated with the Off_PE_Index.
`One_Structure`	Basic data structure corresponding to the rows in the index array. The index array can be thought of as a "Many of One" relationship (e.g. Many Faces of Each Cell, or Faces_of_Cells), with each row of the array signifying all the "Many" items that correspond to that "One" row.
`Trace`	The trace for the communication associated with the index.

The Data_Index Class code listing in § D.4 on page 339 contains additional documentation. The Data_Index Class also contains a Unit Test Program which is listed in § D.4.9 on page 360.

9.4.1 Initialize_Data_Index Procedure

The Initialize_Data_Index procedure allocates and initializes a Data_Index object.

Calling syntax:

```
call Initialize (Index, Many_Structure, One_Structure, Many_of_One_Vector, Many_of_One_Array, Many_of_One_Ragged, status)
```

Input variables:

`Index`	The Data_Index object to be initialized.
`Many_of_One_Array`	The index vector, giving indices for the "Many" that correspond with each "One". Only the rows ("Ones") for this PE are included. Zero entries signify the lack of a reference. If this is specified, then Many_of_One_Vector and Many_of_One_Ragged should not be specified. [Optional]
`Many_of_One_Ragged`	The index vector, giving indices for the "Many" that correspond with each "One". Only the rows ("Ones") for this PE are included. Zero entries signify the lack of a reference. If this is specified, then Many_of_One_Vector and Many_of_One_Array should not be specified. [Optional] – NOT IMPLEMENTED YET.
`Many_of_One_Vector`	The index vector, giving indices for the "Many" that correspond with each "One". Only the rows ("Ones") for this PE are included. Zero entries signify the lack of a reference. If this is specified, then Many_of_One_Array and Many_of_One_Ragged should not be specified. [Optional]
`Many_Structure`	A data structure that corresponds to the columns in the index array.
`One_Structure`	A data structure that corresponds to the rows in the index array.

Output variables:

`Index`	The Data_Index object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.

The Initialize_Data_Index code listing in § D.4.1 on page 342 contains additional documentation.

9.4.2 Finalize_Data_Index Procedure

The Finalize_Data_Index procedure deallocates and finalizes a Data_Index object.

Calling syntax:

```
call Finalize (Index, status)
```

Input variables:

`Index`	The Data_Index object to be finalized.

Output variables:

`Index`	The Data_Index object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Data_Index code listing in § D.4.2 on page 346 contains additional documentation.

9.4.3 Valid_State_Data_Index Procedure

The Valid_State_Data_Index procedure returns true iff the Data_Index is in a valid state – that is, iff the Data_Index passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Index)
```

Input variables:

`Index`	The Data_Index to be checked.

Output variable:

`Valid_State`	True iff the Data_Index is in a valid state.

The Valid_State_Data_Index code listing in § D.4.3 on page 348 contains additional documentation.

9.4.4 Initialized_Data_Index Procedure

The Initialized_Data_Index procedure returns true iff the Data_Index object has been initialized.

Calling syntax:

```
Logical = Initialized(Index)
```

Input variable:

`Index`	The Data_Index object to be examined.

Output variable:

`Initialized`	True iff the Data_Index object has been initialized.

The Initialized_Data_Index code listing in § D.4.4 on page 349 contains additional documentation.

9.4.5 Generate_Shell_Partition Procedure

The Generate_Shell_Partition procedure returns the cell numbering for a "Shell Partitioning". That is, it returns an array giving cell number as a function of i, j, and k. It is called internally by the Initialize_Shell_-Partition procedure in section 9.4.7.

The idea behind a "Shell Partitioning" is to develop a numbering for a *very bad* partitioning. This can then be used to test algorithms to see how well they behave under adverse situations.

All the Shell Partitionings correspond to a structured, "linear", "square" or "cubic", mesh. The total size of the mesh is $\mathrm{Size}^{\mathrm{NDimensions}}$, where Size is any integer (and is equal to the number of processors) and NDimensions is 1, 2 or 3. Each processor then gets

$$\mathrm{PE}^{\mathrm{NDimensions}} - (\mathrm{PE}-1)^{\mathrm{NDimensions}} \tag{9.1}$$

cells, where PE is the processor number. The following is a conceptual description of what is happening. In 1-D, each PE gets one cell – an even partitioning. In 2-D each PE gets the newly added last column and last row of a square, for example

```
XXXXX
    X
    X
    X
    X
```

for processor 5. In 3-D, each PE gets the newly added top and two sides (three sides total) of a cube.

This routine provides a functional relationship for cell numbers by (i,j,k) triplet. The cell numbers are contiguous for a PE, e.g. in 3-D the first PE has number 1, the second PE has numbers 2-8, the third PE has numbers 9-27, etc.

Calling syntax:

```
call Generate_Shell_Partition  (c, i_of_c, j_of_c, k_of_c,
                                NDimensions, NNodes_per_Side, Output)
```

Input variables:

`NDimensions`	The number of dimensions (1=line, 2=square, 3=cube).
`NNodes_per_Side`	The number of nodes in the line, or on the edges of the square or cube.
`Output`	Output toggle.

Output variables:

`c`	The cell numbers as a function of i, j, and k.
`i_of_c, j_of_c, k_of_c`	The i, j, and k numbers for a particular cell number.

The Generate_Shell_Partition code listing in § D.4.5 on page 350 contains additional documentation.

9.4.6 Get_Values_Data_Index Procedure

The Get_Values_Data_Index procedure gets the index values from a Data_Index object.

Calling syntax:

```
Values = Index                    or
call Get_Values (Values, Index)
```

Input variable:

`Index`	The Data_Index object to be queried.

Output variable:

`Values`	The index values from the Data_Index object.

The Get_Values_Data_Index code listing in § D.4.6 on page 352 contains additional documentation.

9.4.7 Initialize_Shell_Partition Procedure

The Initialize_Shell_Partition procedure sets up the Cell_Structure, the Node_Structure, and the Nodes_of_Cells_Index for a "Shell Partitioning". The "Shell Partitioning" numbering scheme, which is designed to be a poorly distributed scheme for testing purposes, is described in the Generate_Shell_Partition procedure in section 9.4.5.

Calling syntax:

```
call Initialize_Shell_Partition  NDimensions, Cell_Structure, Node_Structure,
                                 Nodes_of_Cells_Index, Output
```

Input variables:

`NDimensions`	The number of dimensions (1=line, 2=square, 3=cube).
`Output`	Output toggle.

Output variables:

`Cell_Structure`	The Cell Base_Structure object.
`Node_Structure`	The Node Base_Structure object.
`Nodes_of_Cells_Index`	The Nodes_of_Cells Data_Index object.

The Initialize_Shell_Partition code listing in § D.4.7 on page 353 contains additional documentation.

9.4.8 Output_Data_Index Procedure

The Output_Data_Index procedure writes out a section of a Data Index to the specified unit.

Calling syntax:

```
call Output (Index, First, Last, Unit, Indent, Output_OPE)
```

Input variables:

`Index`	The Data_Index object to be queried.
`First`	The first location to be output. [Optional]
`Last`	The last location to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]
`Indent`	Number of indentation characters. [Optional]
`Output_OPE`	Toggle for outputting the Off_PE_Index information. [Optional]

The Output_Data_Index code listing in § D.4.8 on page 356 contains additional documentation.

9.5 Assembled_Vector Class

The Assembled_Vector Class is used to describe an assembled vector (existing only on a single processor) in the CÆSAR Code Package. An Assembled_Vector is part of the overall data structure strategy in CÆSAR, which is made up of the following classes: Base_Structure, Data_Index, Assembled_Vector, Distributed_Vector, Overlapped_Vector, and Collected_Array. A description of the overall data structure strategy can be found in the Data_Structures Module (described in chapter 9 on page 63).

Assembled_Vector public procedures:

Fundamental procedures	
`Initialize`	Initializes an Assembled_Vector object.
`Finalize`	Finalizes an Assembled_Vector object.
`Valid_State`	Returns false iff an Assembled_Vector object is in an invalid state.
`Initialized`	Returns true iff an Assembled_Vector object has been initialized.
Operations	
`Get_Values`	Gets the values from an Assembled_Vector object and returns them in a bare naked vector (also has an assignment interface).
`Locus`	Returns the locus of the Assembled_Vector object.
`Name`	Returns the name of the Assembled_Vector object.
`Output`	Writes out the Assembled_Vector object.
`Set_Values`	Sets the values of the Assembled_Vector object to a bare naked vector (also has an assignment interface).
`Set_Version`	Sets the version number of the Assembled_Vector object (also has an assignment interface).
`Version`	Returns the version number of the Assembled_Vector object.

Assembled_Vector public defined type:

Assembled_Vector type	
`Dimensionality`	The number of dimensions that the "vector" has, including the dimension that is spread over the processors. "Ragged_Right" indices are signified by a Dimensionality of -1. (Ragged_Right is not yet implemented.)
`Dimensions`	The extents of the dimensions that the "vector" has, including the dimension that is spread over the processors, which is last.
`Initialized`	Initialization status.
`Name`	The name for this variable (especially useful in a vector of Assembled Vectors).
`NValues_Total`	Total number of values in this vector.
`Structure`	Basic data structure for the axis that is spread over the processors.
`Values`$\{n\}$	Values in the vector, only defined on the IO PE. Values may have either 1, 2, 3, or 4 dimensions (n = 1, 2, 3, or 4), or be a ragged right array (n = RR). The last dimension is always the dimension to be spread across the processors. Only one of the variables will be allocated for a given object. Ragged right arrays have not been implemented yet.
`Version`	Version number which is incremented every time the vector is modified, or is synced with the version number of a data structure that it depends on when it is updated.

The Assembled_Vector Class code listing in § D.5 on page 361 contains additional documentation. The Assembled_Vector Class also contains a Unit Test Program which is listed in § D.5.12 on page 376.

9.5.1 Initialize_Assembled_Vector Procedure

The Initialize_Assembled_Vector procedure allocates and initializes a Assembled_Vector object.

Calling syntax:

```
call Initialize (AV, Structure, Dimensionality, Name, status, dim1, dim2, dim3)
```

Input variables:

`AV`	The Assembled_Vector object to be initialized.
`Structure`	The basic data structure for this Assembled_Vector.
`Dimensionality`	The number of dimensions that the "vector" has, including the dimension that is spread over the processors. "Ragged_Right" vectors are signified by a Dimensionality of -1.
`Name`	The name for this variable (especially useful in a vector of Assembled Vectors). [Optional]
`dim`$\{n\}$	The dimensions for this "vector". There must be dimensions specified up to a number one less than the Dimensionality. [Optional]

Output variables:

`AV`	The Assembled_Vector object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`Length_PE`	Length on this PE.

The Initialize_Assembled_Vector code listing in § D.5.1 on page 364 contains additional documentation.

9.5.2 Finalize_Assembled_Vector Procedure

The Finalize_Assembled_Vector procedure deallocates and finalizes an Assembled_Vector object.

Calling syntax:

```
call Finalize (AV, status)
```

Input variables:

`AV`	The Assembled_Vector object to be finalized.

Output variables:

`AV`	The Assembled_Vector object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Assembled_Vector code listing in § D.5.2 on page 366 contains additional documentation.

9.5.3 Valid_State_Assembled_Vector Procedure

The Valid_State_Assembled_Vector procedure returns true iff the Assembled_Vector is in a valid state – that is, iff the Assembled_Vector passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(AV)
```

Input variables:

`AV`	The Assembled_Vector to be checked.

Output variable:

`Valid_State`	True iff the Assembled_Vector is in a valid state.

The Valid_State_Assembled_Vector code listing in § D.5.3 on page 368 contains additional documentation.

9.5.4 Initialized_Assembled_Vector Procedure

The Initialized_Assembled_Vector procedure returns true iff the Assembled_Vector object has been initialized.

Calling syntax:

```
Logical = Initialized(AV)
```

Input variable:

`AV`	The Assembled_Vector object to be examined.

Output variable:

`Initialized`	True iff the Assembled_Vector object has been initialized.

The Initialized_Assembled_Vector code listing in § D.5.4 on page 369 contains additional documentation.

9.5.5 Get_Locus_Assembled_Vector Procedure

The Get_Locus_Assembled_Vector procedure returns the locus of the Assembled Vector.

Calling syntax:

```
Character = Locus(AV)
```

Input variable:

`AV`	The Assembled_Vector object to be queried.

Output variable:

`Locus`	The locus of the Assembled_Vector object.

The Get_Locus_Assembled_Vector code listing in § D.5.5 on page 370 contains additional documentation.

9.5.6 Get_Name_Assembled_Vector Procedure

The Get_Name_Assembled_Vector procedure returns the name of the Assembled Vector.

Calling syntax:

```
Character = Name(AV)
```

Input variable:

`AV`	The Assembled_Vector object to be queried.

Output variable:

`Name`	The name of the Assembled_Vector object.

The Get_Name_Assembled_Vector code listing in § D.5.6 on page 370 contains additional documentation.

9.5.7 Get_Values_Assembled_Vector Procedure

The Get_Values_Assembled_Vector procedure gets the values from an Assembled Vector.

Calling syntax:

```
Values = AV                    or
call Get_Values (Values, AV)
```

Input variable:

`AV`	The Assembled_Vector object to be queried.

Output variable:

`Values`	The bare naked vector of values from the Assembled_Vector object, only defined on the IO PE.

The Get_Values_Assembled_Vector code listing in § D.5.7 on page 371 contains additional documentation.

9.5.8 Get_Version_Assembled_Vector Procedure

The Get_Version_Assembled_Vector procedure returns the version number for the Assembled Vector.

Calling syntax:

```
Integer = Version(AV)
```

Input variables:

`AV`	The Assembled_Vector object to be queried.

Output variable:

`Version`	The version number of the Assembled_Vector object.

The Get_Version_Assembled_Vector code listing in § D.5.8 on page 372 contains additional documentation.

9.5.9 Output_Assembled_Vector Procedure

The Output_Assembled_Vector procedure writes out a section of an Assembled Vector to the specified unit.

Calling syntax:

```
call Output (AV, First, Last, Unit)
```

Input variables:

`AV`	The Assembled_Vector object to be queried.
`First`	The first location to be output. [Optional]
`Last`	The last location to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]

The Output_Assembled_Vector code listing in § D.5.9 on page 373 contains additional documentation.

9.5.10 Set_Values_Assembled_Vector Procedure

The Set_Values_Assembled_Vector procedure sets the values for the Assembled Vector.

Calling syntax:

```
AV = Values                      or
call Set_Values (AV, Values)
```

Input variable:

`Values`	The bare naked vector of values for the Assembled_Vector object, only defined on the IO PE.

Input/Output variable:

`AV`	The Assembled_Vector object to be set.

Internal variable:

`Version_Increment`	The amount that the version number is incremented, which is a global class variable.

The Set_Values_Assembled_Vector code listing in § D.5.10 on page 375 contains additional documentation.

9.5.11 Set_Version_Assembled_Vector Procedure

The Set_Version_Assembled_Vector procedure sets the version number for the Assembled Vector.

Calling syntax:

```
AV = Version                        or
call Set_Version (AV, Version)
```

Input variable:

`Version`	The version number for the Assembled_Vector object.

Input/Output variable:

`AV`	The Assembled_Vector object to be set.

The Set_Version_Assembled_Vector code listing in § D.5.11 on page 376 contains additional documentation.

9.6 Distributed_Vector Class

The Distributed_Vector Class is used to describe a distributed vector (existing across all the processors) in the Cæsar Code Package. A Distributed_Vector is part of the overall data structure strategy in Cæsar, which is made up of the following classes: Base_Structure, Data_Index, Assembled_Vector, Distributed_Vector, Overlapped_Vector, and Collected_Array. A description of the overall data structure strategy can be found in the Data_Structures Module (described in chapter 9 on page 63).

The form of a Distributed_Vector object is given by:

```
Vector ( [dim1, [dim2, [dim3,]]] Structure_Axis )
```

or

```
Vector ( dim_ragged_right, Structure_Axis )
--> not implemented
```

where Structure_Axis refers to the axis which is spread across the processors.

Distributed_Vector public procedures:

Fundamental procedures	
`Initialize`	Initializes a Distributed_Vector object.
`Finalize`	Finalizes a Distributed_Vector object.
`Valid_State`	Returns false iff a Distributed_Vector object is in an invalid state.
`Initialized`	Returns true iff a Distributed_Vector object has been initialized.
Operations	
`Assemble`	Sets an Assembled_Vector object to a Distributed_Vector object by assembling the data on the IO PE (also has an assignment interface).
`Distribute`	Sets a Distributed_Vector object to an Assembled_Vector object by distributing the data from the IO PE to all the PEs (also has an assignment interface).
`Get_Values`	Gets the values from a Distributed_Vector object and returns them in a bare naked vector (also has an assignment interface).
`Locus`	Returns the locus of the Distributed_Vector object.
`Name`	Returns the name of the Distributed_Vector object.
`Output`	Writes out the Distributed_Vector object.
`Set_Values`	Sets the values of the Distributed_Vector object to a bare naked vector (also has an assignment interface).
`Set_Version`	Sets the version number of the Distributed_Vector object (also has an assignment interface).
`Version`	Returns the version number of the Distributed_Vector object.

Distributed_Vector public defined type:

Distributed_Vector type

`Dimensionality`	The number of dimensions that the index"vector" has, including the dimension that is spread over the processors. "Ragged_Right" indices are signified by a Dimensionality of -1. (Ragged_Right is not yet implemented.)
`Dimensions`	The extents of the dimensions that the "vector" has, including the dimension that is spread over the processors, which is last.
`Initialized`	Initialization status.
`Name`	The name for this variable (especially useful in a vector of Distributed Vectors).
`NValues_PE`	Number of values on this PE.
`NValues_Total`	Total number of values in the entire vector (including all PEs).
`NValues_Vector`	A vector containing the number of values on each PE.
`Structure`	Basic data structure for the axis that is spread over the processors.
`Values`$\{n\}$	Values in the vector, with a different length on each PE. Values may have either 1, 2, 3, or 4 dimensions (n = 1, 2, 3, or 4), or be a ragged right array (n = RR). The last dimension is always the dimension to be spread across the processors. Only one of the variables will be allocated for a given object. Ragged right arrays have not been implemented yet.
`Version`	Version number which is incremented every time the vector is modified, or is synced with the version number of a data structure that it depends on when it is updated.

The Distributed_Vector Class code listing in § D.6 on page 378 contains additional documentation. The Distributed_Vector Class also contains a Unit Test Program which is listed in § D.6.14 on page 399.

9.6.1 Initialize_Distributed_Vector Procedure

The Initialize_Distributed_Vector procedure allocates and initializes a Distributed_Vector object.

Calling syntax:

```
call Initialize (DV, Structure, Dimensionality, Name, status, dim1, dim2, dim3)
```

Input variables:

`DV`	The Distributed_Vector object to be initialized.
`Structure`	The basic data structure for this Distributed_Vector.
`Dimensionality`	The number of dimensions that the "vector" has, including the dimension that is spread over the processors. "Ragged_Right" vectors are signified by a Dimensionality of -1.
`Name`	The name for this variable (especially useful in a vector of Distributed Vectors). [Optional]
`dim`$\{n\}$	The dimensions for this "vector". There must be dimensions specified up to a number one less than the Dimensionality. [Optional]

Output variables:

`DV`	The Distributed_Vector object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.
`NSlice`	Number of values that are on one slice, given by a constant location on the distributed axis.

The Initialize_Distributed_Vector code listing in § D.6.1 on page 382 contains additional documentation.

9.6.2 Finalize_Distributed_Vector Procedure

The Finalize_Distributed_Vector procedure deallocates and finalizes an Distributed_Vector object.

Calling syntax:

```
call Finalize (DV, status)
```

Input variables:

`DV`	The Distributed_Vector object to be finalized.

Output variables:

`DV`	The Distributed_Vector object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Distributed_Vector code listing in § D.6.2 on page 384 contains additional documentation.

9.6.3 Valid_State_Distributed_Vector Procedure

The Valid_State_Distributed_Vector procedure returns true iff the Distributed_Vector is in a valid state – that is, iff the Distributed_Vector passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(DV)
```

Input variables:

`DV`	The Distributed_Vector to be checked.

Output variable:

`Valid_State`	True iff the Distributed_Vector is in a valid state.

The Valid_State_Distributed_Vector code listing in § D.6.3 on page 386 contains additional documentation.

9.6.4 Initialized_Distributed_Vector Procedure

The Initialized_Distributed_Vector procedure returns true iff the Distributed_Vector object has been initialized.

Calling syntax:

```
Logical = Initialized(DV)
```

Input variable:

`DV`	The Distributed_Vector object to be examined.

Output variable:

`Initialized`	True iff the Distributed_Vector object has been initialized.

The Initialized_Distributed_Vector code listing in § D.6.4 on page 387 contains additional documentation.

9.6.5 Assemble_AV_from_DV Procedure

The Assemble_AV_from_DV procedure sets an Assembled Vector to a Distributed Vector by assembling the data on the IO PE.

Calling syntax:

```
AV = DV                    or
call Assemble (AV, DV)
```

Input variable:

`DV`	The Distributed_Vector object to be assembled.

Output variable:

`AV`	The Assembled_Vector object, completely on the IO PE.

The Assemble_AV_from_DV code listing in § D.6.5 on page 388 contains additional documentation.

9.6.6 Distribute_AV_to_DV Procedure

The Distribute_AV_to_DV procedure sets a Distributed Vector to an Assembled Vector by distributing the data from the IO PE to all the PEs.

Calling syntax:

```
DV = AV                      or
call Distribute (DV, AV)
```

Input variable:

`AV`	The Assembled_Vector object to be distributed.

Output variable:

`DV` The Distributed_Vector object, distributed over all the PEs.

The Distribute_AV_to_DV code listing in § D.6.6 on page 389 contains additional documentation.

9.6.7 Get_Locus_Distributed_Vector Procedure

The Get_Locus_Distributed_Vector procedure returns the locus of the Distributed Vector.

Calling syntax:

```
Character = Locus(DV)
```

Input variable:

`DV` The Distributed_Vector object to be queried.

Output variable:

`Locus` The locus of the Distributed_Vector object.

The Get_Locus_Distributed_Vector code listing in § D.6.7 on page 391 contains additional documentation.

9.6.8 Get_Name_Distributed_Vector Procedure

The Get_Name_Distributed_Vector procedure returns the name of the Distributed Vector.

Calling syntax:

```
Character = Name(DV)
```

Input variable:

`DV` The Distributed_Vector object to be queried.

Output variable:

`Name` The name of the Distributed_Vector object.

The Get_Name_Distributed_Vector code listing in § D.6.8 on page 391 contains additional documentation.

9.6.9 Get_Values_Distributed_Vector Procedure

The Get_Values_Distributed_Vector procedure gets the values from a Distributed Vector.

Calling syntax:

```
Values = DV                     or
call Get_Values (Values, DV)
```

Input variable:

`DV` The Distributed_Vector object to be queried.

Output variable:

`Values`	The bare naked vector of values from the Distributed_Vector object, defined differently on each PE.

The Get_Values_Distributed_Vector code listing in § D.6.9 on page 392 contains additional documentation.

9.6.10 Get_Version_Distributed_Vector Procedure

The Get_Version_Distributed_Vector procedure returns the version number for the Distributed Vector.

Calling syntax:

```
Integer = Version(DV)
```

Input variables:

`DV`	The Distributed_Vector object to be queried.

Output variable:

`Version`	The version number of the Distributed_Vector object.

The Get_Version_Distributed_Vector code listing in § D.6.10 on page 393 contains additional documentation.

9.6.11 Output_Distributed_Vector Procedure

The Output_Distributed_Vector procedure writes out a section of a Distributed Vector to the specified unit.

Calling syntax:

```
call Output (DV, First, Last, Unit, Indent)
```

Input variables:

`DV`	The Distributed_Vector object to be queried.
`First`	The first location to be output. [Optional]
`Last`	The last location to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]
`Indent`	Number of indentation characters. [Optional]

The Output_Distributed_Vector code listing in § D.6.11 on page 393 contains additional documentation.

9.6.12 Set_Values_Distributed_Vector Procedure

The Set_Values_Distributed_Vector procedure sets the values for the Distributed Vector.

Calling syntax:

`DV = Values` or
`call Set_Values (DV, Values)`

Input variable:

`Values` The bare naked vector of values for the Distributed_Vector object, defined differently on each PE.

Input/Output variable:

`DV` The Distributed_Vector object to be set.

Internal variable:

`Version_Increment` The amount that the version number is incremented, which is a global class variable.

The Set_Values_Distributed_Vector code listing in § D.6.12 on page 398 contains additional documentation.

9.6.13 Set_Version_Distributed_Vector Procedure

The Set_Version_Distributed_Vector procedure sets the version number for the Distributed Vector.

Calling syntax:

```
DV = Version                          or
call Set_Version (DV, Version)
```

Input variable:

`Version` The version number for the Distributed_Vector object.

Input/Output variable:

`DV` The Distributed_Vector object to be set.

The Set_Version_Distributed_Vector code listing in § D.6.13 on page 399 contains additional documentation.

9.7 Overlapped_Vector Class

The Overlapped_Vector Class is used to describe an overlapped vector (existing across all the processors) in the Cæsar Code Package. An Overlapped_Vector is part of the overall data structure strategy in Cæsar, which is made up of the following classes: Base_Structure, Data_Index, Assembled_Vector, Distributed_Vector, Overlapped_Vector, and Collected_Array. A description of the overall data structure strategy can be found in the Data_Structures Module (described in chapter 9 on page 63).

The form of a collected Overlapped_Vector object is given by:

```
Array ( [dim1, [dim2, [dim3,]]] One_Axis [, Many_Axis] )
```

or

```
Array ( dim_ragged_right, One_Axis [, Many_Axis] )
--> not implemented
```

where One_Axis refers to the axis which is spread across the processors.

Overlapped_Vector public procedures:

Fundamental procedures

`Initialize`	Initializes an Overlapped_Vector object.
`Finalize`	Finalizes an Overlapped_Vector object.
`Valid_State`	Returns false iff an Overlapped_Vector object is in an invalid state.
`Initialized`	Returns true iff an Overlapped_Vector object has been initialized.
Operations	
`Collect_and_Access`	Another name for Get_Values.
`Collect_and_Combine`	Collects the values from an Overlapped Vector, and then combines them to form a Distributed Vector, according to the internal Many_of_One_Index object. The resultant Distributed Vector is distributed according to the One Structure of the Overlapped Vector. A combination operator, to be put in the place of "Combine", must be specified. Allowed values for "Combine" are: `Average`, `MAX`, `MIN`, or `SUM`. Collect_and_SUM also has an assignment interface.
`Gather`	Does the communication necessary to set an Overlapped Vector from a Distributed Vector (also has an assignment interface).
`Get_Values`	Collects and accesses the values from an Overlapped_Vector object and returns them in a bare naked array (also has an assignment interface).
`Many_Locus`	Returns the Many Structure locus of the Overlapped_Vector object.
`Name`	Returns the name of the Overlapped_Vector object.
`One_Locus`	Returns the One Structure locus of the Overlapped_Vector object.
`Output`	Writes out the Overlapped_Vector object.
`Set_Version`	Sets the version number of the Overlapped_Vector object (also has an assignment interface).
`Version`	Returns the version number of the Overlapped_Vector object.

Overlapped_Vector public defined type:

Overlapped_Vector type	
`Dimensionality`	The number of dimensions that the "vector" has, including the dimension that is spread over the processors. "Ragged_Right" indices are signified by a Dimensionality of -1. (Ragged_Right is not yet implemented.)
`Dimensions`	The extents of the dimensions that the "vector" has, including the dimension that is spread over the processors, which is last.
`DV`	A pointer to the Distributed Vector that this Overlapped Vector is based on.
`DV_Internal`	An internal Distributed Vector that is constructed if the Overlapped Vector is not based on an external Distributed Vector.
`Initialized`	Initialization status.
`Many_of_One_Index`	The Index that is used to modify the Distributed Vector.
`Many_Structure`	Basic data structure which corresponds to the structure of the Distributed Vector that this Overlapped Vector is based on.
`Name`	The name for this variable (especially useful in a vector of Overlapped Vectors).
`One_Structure`	Basic data structure which corresponds to the way that this Overlapped Vector has been formed. If this Overlapped Vector were to be combined, it would result in a Distributed Vector with a One_Structure basis.
`Overlap_Index`	The index for the distributed axis of the off-PE values.
`Overlap_Trace`	The trace for the distributed axis of the off-PE values.
`Overlap_Values`$\{n\}$	Off-PE values in the vector, that are stored locally, with a different length on each PE. Values may have either 1, 2, 3, or 4 dimensions (n = 1, 2, 3, or 4), or be a ragged right array (n = RR). The last dimension is always the dimension to be spread across the processors. Only one of the variables will be allocated for a given object. Ragged right arrays have not been implemented yet.
`Version`	Version number which is incremented every time the vector is modified, or is synced with the version number of a data structure that it depends on when it is updated.

The Overlapped_Vector Class code listing in § D.7 on page 401 contains additional documentation. The Overlapped_Vector Class also contains a Unit Test Program which is listed in § D.7.13 on page 432.

9.7.1 Initialize_Overlapped_Vector Procedure

The Initialize_Overlapped_Vector procedure allocates and initializes a Overlapped_Vector object. There are two ways to initialize an Overlapped_Vector object, depending on whether or not the underlying Distributed_-Vector object is internally created.

Calling syntax:

```
call Initialize (OV, DV, Many_of_One_Index, Name, status)
```
or
```
call Initialize (OV, Many_of_One_Index, Dimensionality, Name, status, dim1, dim2, dim3)
```

Input variables:

`OV`	The Overlapped_Vector object to be initialized.
`DV`	The Distributed_Vector object which the Overlapped_Vector is to be based on.
`Many_of_One_Index`	An index giving the relationship of the "Many" and "One" axes to each other for this Overlapped_Vector.
`Dimensionality`	The number of dimensions that the "vector" has, including the dimension that is spread over the processors. "Ragged_Right" vectors are signified by a Dimensionality of -1.
`Name`	The name for this variable (especially useful in a vector of Overlapped Vectors). [Optional]
`dim`$\{n\}$	The dimensions for this "vector". There must be dimensions specified up to a number one less than the Dimensionality. These are only needed in the second form of the call. [Optional]

Output variables:

`OV`	The Overlapped_Vector object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Overlapped_Vector code listing in § D.7.1 on page 406 contains additional documentation.

9.7.2 Finalize_Overlapped_Vector Procedure

The Finalize_Overlapped_Vector procedure deallocates and finalizes an Overlapped_Vector object.

Calling syntax:

```
call Finalize (OV, status)
```

Input variables:

`OV`	The Overlapped_Vector object to be finalized.

Output variables:

`OV`	The Overlapped_Vector object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Overlapped_Vector code listing in § D.7.2 on page 410 contains additional documentation.

9.7.3 Valid_State_Overlapped_Vector Procedure

The Valid_State_Overlapped_Vector procedure returns true iff the Overlapped_Vector is in a valid state – that is, iff the Overlapped_Vector passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(OV)
```

Input variables:

`OV`	The Overlapped_Vector to be checked.

Output variable:

`Valid_State`	True iff the Overlapped_Vector is in a valid state.

The Valid_State_Overlapped_Vector code listing in § D.7.3 on page 411 contains additional documentation.

9.7.4 Initialized_Overlapped_Vector Procedure

The Initialized_Overlapped_Vector procedure returns true iff the Overlapped_Vector object has been initialized.

Calling syntax:

```
Logical = Initialized(OV)
```

Input variable:

`OV`	The Overlapped_Vector object to be examined.

Output variable:

`Initialized`	True iff the Overlapped_Vector object has been initialized.

The Initialized_Overlapped_Vector code listing in § D.7.4 on page 413 contains additional documentation.

9.7.5 Collect_and_Combine_DV_from_OV Procedure

The Collect_and_Combine_DV_from_OV procedures collect the values from an Overlapped Vector, and then combine them to form a Distributed Vector. Specifically, each procedure does a Collect and Combine on the data in the Overlapped Vector, with the same result that would occur if an intermediate Collected Array had been constructed. (Collect_and_Conserve has not been implemented yet.)

Calling syntax:

```
call Collect_and_Average (DV, OV)     ,
call Collect_and_Conserve (DV, OV)    ,
call Collect_and_MAX (DV, OV)         ,
call Collect_and_MIN (DV, OV)         ,
call Collect_and_SUM (DV, OV)         or
DV = OV [Collect_and_SUM]
```

Input variable:

`OV` The Overlapped_Vector object to be queried.

Output variable:

`DV` The Distributed_Vector object result, distributed with the One_Structure of the Overlapped_Vector object.

The Collect_and_Combine_DV_from_OV code listing in § D.7.5 on page 414 contains additional documentation.

9.7.6 Gather_OV_from_DV Procedure

The Gather_OV_from_DV procedure does the communication necessary to set an Overlapped Vector from a Distributed Vector. No communication is done if the Overlapped Vector is already up-to-date with the Distributed Vector.

Calling syntax:

```
OV = DV                 or
call Gather (OV, DV)
```

Input variable:

`DV` The Distributed_Vector object to be gathered.

Output variable:

`OV` The Overlapped_Vector object, updated to correspond with the input Distributed_-Vector object.

The Gather_OV_from_DV code listing in § D.7.6 on page 419 contains additional documentation.

9.7.7 Get_Locus_Overlapped_Vector Procedure

The Get_Many_Locus_OV and Get_One_Locus_OV procedures return the loci of the Many and One Structures of the Overlapped Vector respectively.

Calling syntax:

```
Character = Many_Locus(OV)    or
Character = One_Locus(OV)
```

Input variable:

`OV` The Overlapped_Vector object to be queried.

Output variable:

`Locus` The specified locus of the Overlapped_Vector object.

The Get_Locus_Overlapped_Vector code listing in § D.7.7 on page 420 contains additional documentation.

9.7.8 Get_Name_Overlapped_Vector Procedure

The Get_Name_Overlapped_Vector procedure returns the name of the Overlapped Vector.

Calling syntax:

```
Character = Name(OV)
```

Input variable:

`OV` The Overlapped_Vector object to be queried.

Output variable:

`Name` The name of the Overlapped_Vector object.

The Get_Name_Overlapped_Vector code listing in § D.7.8 on page 421 contains additional documentation.

9.7.9 Get_Values_Overlapped_Vector Procedure

The Get_Values_Overlapped_Vector procedure gets the values from an Overlapped Vector, in the form of a Bare Naked Array. Specifically, it does a Collect and Access on the data in the Overlapped Vector, with the same result that would occur if an intermediate Collected Array had been constructed.

Calling syntax:

```
Values = OV                              ,
call Get_Values (Values, OV)             or
call Collect_and_Access (Values, OV)
```

Input variable:

`OV` The Overlapped_Vector object to be queried.

Output variable:

`Values`	The bare naked array of values from the Overlapped_Vector object, defined differently on each PE.

The Get_Values_Overlapped_Vector code listing in § D.7.9 on page 422 contains additional documentation.

9.7.10 Get_Version_Overlapped_Vector Procedure

The Get_Version_Overlapped_Vector procedure returns the version number for the Overlapped Vector.

Calling syntax:

```
Integer = Version(OV)
```

Input variables:

`OV`	The Overlapped_Vector object to be queried.

Output variable:

`Version`	The version number of the Overlapped_Vector object.

The Get_Version_Overlapped_Vector code listing in § D.7.10 on page 427 contains additional documentation.

9.7.11 Output_Overlapped_Vector Procedure

The Output_Overlapped_Vector procedure writes out a section of an Overlapped Vector to the specified unit.

Calling syntax:

```
call Output (OV, Many_First, Many_Last, One_First, One_Last, Unit)
```

Input variables:

`OV`	The Overlapped_Vector object to be queried.
`Many_First`	The first location on the Many Axis to be output. [Optional]
`Many_Last`	The last location on the Many Axis to be output. [Optional]
`One_First`	The first location on the One Axis to be output. [Optional]
`One_Last`	The last location on the One Axis to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]

The Output_Overlapped_Vector code listing in § D.7.11 on page 427 contains additional documentation.

9.7.12 Set_Version_Overlapped_Vector Procedure

The Set_Version_Overlapped_Vector procedure sets the version number for the Overlapped Vector.

Calling syntax:

```
OV = Version                          or
call Set_Version (OV, Version)
```

Input variable:

`Version` The version number for the Overlapped_Vector object.

Input/Output variable:

`OV` The Overlapped_Vector object to be set.

The Set_Version_Overlapped_Vector code listing in § D.7.12 on page 431 contains additional documentation.

9.8 Collected_Array Class

The Collected_Array Class is used to describe a collected array (existing across all the processors) in the Cæsar Code Package. A Collected_Array is part of the overall data structure strategy in Cæsar, which is made up of the following classes: Base_Structure, Data_Index, Assembled_Vector, Distributed_Vector, Overlapped_Vector, and Collected_Array. A description of the overall data structure strategy can be found in the Data_Structures Module (described in chapter 9 on page 63).

The form of a Collected_Array object is given by:

```
Array ( [dim1, [dim2, [dim3,]]] One_Axis [, Many_Axis] )
```

or

```
Array ( dim_ragged_right, One_Axis [, Many_Axis] )
--> not implemented
```

where One_Axis refers to the axis which is spread across the processors.

Collected_Array public procedures:

Fundamental procedures	
`Initialize`	Initializes a Collected_Array object.
`Finalize`	Finalizes a Collected_Array object.
`Valid_State`	Returns false iff a Collected_Array object is in an invalid state.
`Initialized`	Returns true iff a Collected_Array object has been initialized.
Operations	
`Collect`	Collects the values from an Overlapped Vector and stores them in a Collected Array.
`Combine_with_Op`	Combines the values from a Collected Array, to form a Distributed Vector. The resultant Distributed Vector is distributed according to the One Structure of the Collected Array. A combination operator, to be put in the place of "Op", must be specified. Allowed values for "Op" are: `Average`, `MAX`, `MIN`, or `SUM`. Combine_with_-SUM also has an assignment interface.
`Gather`	Does the communication necessary to set a Collected Array from a Distributed Vector (also has an assignment interface).
`Gather_and_Collect`	Another name for Gather.
`Get_Values`	Accesses the values from a Collected_Array object and returns them in a Bare Naked Array (also has an assignment interface).
`Many_Locus`	Returns the Many Structure locus of the Collected_Array object.
`Name`	Returns the name of the Collected_Array object.
`One_Locus`	Returns the One Structure locus of the Collected_Array object.
`Output`	Writes out the Collected_Array object.
`Set_Values`	Sets the values of the Collected_Array object to a Bare Naked Array (also has an assignment interface).
`Set_Version`	Sets the version number of the Collected_Array object (also has an assignment interface).
`Version`	Returns the version number of the Collected_Array object.

Collected_Array public defined type:

Collected_Array type	
`A_Dimensionality`	The actual number of dimensions that the "array" has, including the dimension that is spread over the processors (the One_Axis), and also including the Many_Axis, if it is present. "Ragged_Right" indices are signified by a Dimensionality of -1. (Ragged_Right is not yet implemented.)
`Dimensionality`	The number of dimensions that the "array" has, including the dimension that is spread over the processors (the One_Axis), but not including the Many_Axis, if it is present. "Ragged_Right" indices are signified by a Dimensionality of -1. (Ragged_Right is not yet implemented.)
`Dimensions`	The extents of the dimensions that the "array" has, including the dimensions for the One_Axis and the Many_Axis.
`Initialized`	Initialization status.
`Many_of_One_Index`	The Index that is used to translate between the Distributed Vectors.
`Many_Structure`	Basic data structure which corresponds to the structure of the Distributed Vector that this Collected Array is based on.
`Name`	The name for this variable (especially useful in a vector of Collected Arrays).
`One_Structure`	Basic data structure which corresponds to the way that this Collected Array has been formed. If this Collected Array were to be combined, it would result in a Distributed Vector with a One_Structure basis.
`Values`$\{n\}$	Values in the array, that are stored locally, with a different length on each PE. Values may have either 1, 2, 3, 4, or 5 dimensions (n = 1, 2, 3, 4, or 5), or be a ragged right array (n = RR). The last dimension is the dimension that is spread across the processors, if the Many_of_One_Index is a vector index. Otherwise, the penultimate axis will be spread across the processors. Only one of the variables will be allocated for a given object. Ragged right arrays have not been implemented yet.
`Version`	Version number which is incremented every time the array is modified, or is synced with the version number of a data structure that it depends on when it is updated.

The Collected_Array Class code listing in § D.8 on page 436 contains additional documentation. The Collected_Array Class also contains a Unit Test Program which is listed in § D.8.15 on page 465.

9.8.1 Initialize_Collected_Array Procedure

The Initialize_Collected_Array procedure allocates and initializes a Collected_Array object. There are two ways to initialize an Collected_Array object, depending on whether or not the underlying Distributed_Vector object is internally created.

Calling syntax:

```
call Initialize (CA, OV, Name, status)
```
or
```
call Initialize (CA, Many_of_One_Index, Dimensionality, Name, status, dim1, dim2, dim3)
```

Input variables:

`CA`	The Collected_Array object to be initialized.
`OV`	An Overlapped_Vector object which the Collected_Array is to be based on. Note that the Collected_Array will not *contain* the Overlapped_Vector or require the Overlapped_Vector to be used in an assignment statement, but rather the Collected_Array will have a structure that is compatible with the Overlapped_Vector and will be set to the Overlapped_Vector during initialization.
`Many_of_One_Index`	An index giving the relationship of the "Many" and "One" axes to each other for this Collected_Array.

`Dimensionality`	The number of dimensions that the “array” has, including the dimension that is spread over the processors (the One_Axis), but not including the Many_Axis, if it is present. “Ragged_Right” vectors are signified by a Dimensionality of -1.
`Name`	The name for this variable (especially useful in a vector of Collected Arrays). [Optional]
`dim`$\{n\}$	The dimensions for this “array”. There must be dimensions specified up to a number one less than the Dimensionality. These are only needed in the second form of the call. [Optional]

Output variables:

`CA`	The Collected_Array object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Collected_Array code listing in § D.8.1 on page 440 contains additional documentation.

9.8.2 Finalize_Collected_Array Procedure

The Finalize_Collected_Array procedure deallocates and finalizes an Collected_Array object.

Calling syntax:

```
call Finalize (CA, status)
```

Input variables:

`CA`	The Collected_Array object to be finalized.

Output variables:

`CA`	The Collected_Array object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Collected_Array code listing in § D.8.2 on page 445 contains additional documentation.

9.8.3 Valid_State_Collected_Array Procedure

The Valid_State_Collected_Array procedure returns true iff the Collected_Array is in a valid state – that is, iff the Collected_Array passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(CA)
```

Input variables:

`CA` The Collected_Array to be checked.

Output variable:

`Valid_State` True iff the Collected_Array is in a valid state.

The Valid_State_Collected_Array code listing in § D.8.3 on page 447 contains additional documentation.

9.8.4 Initialized_Collected_Array Procedure

The Initialized_Collected_Array procedure returns true iff the Collected_Array object has been initialized.

Calling syntax:

```
Logical = Initialized(CA)
```

Input variable:

`CA` The Collected_Array object to be examined.

Output variable:

`Initialized` True iff the Collected_Array object has been initialized.

The Initialized_Collected_Array code listing in § D.8.4 on page 449 contains additional documentation.

9.8.5 Collect_CA_from_OV Procedure

The Collect_CA_from_OV procedure collects the values from an Overlapped Vector to form a Collected Array.

Calling syntax:

```
call Collect (CA, OV)    or
CA = OV
```

Input variable:

`OV` The Overlapped_Vector object to be queried.

Output variable:

`CA` The Collected_Array object result.

The Collect_CA_from_OV code listing in § D.8.5 on page 449 contains additional documentation.

9.8.6 Combine_DV_from_CA Procedure

The Combine_DV_from_CA procedures combine the values from a Collected Array to form a Distributed Vector. (Conserve has not been implemented yet.)

Calling syntax:

```
call Combine_with_Average (DV, CA)     ,
call Combine_with_Conserve (DV, CA)    ,
call Combine_with_MAX (DV, CA)         ,
call Combine_with_MIN (DV, CA)         ,
call Combine_with_SUM (DV, CA)         or
DV = CA [SUM]
```

Input variable:

`CA` The Collected_Array object to be queried.

Output variable:

`DV` The Distributed_Vector object result, distributed with the One_Structure of the Collected_Array object.

The Combine_DV_from_CA code listing in § D.8.6 on page 450 contains additional documentation.

9.8.7 Gather_and_Collect_CA_from_DV Procedure

The Gather_and_Collect_CA_from_DV procedure does the communication necessary to set a Collected Array from a Distributed Vector, according to the Many_of_One_Index which is inside the Collected Array. No communication is done if the Collected Array is already up-to-date with the Distributed Vector.

Calling syntax:

```
CA = DV                                ,
call Gather (CA, DV)                   or
call Gather_and_Collect (CA, DV)
```

Input variable:

`DV` The Distributed_Vector object to be gathered.

Output variable:

`CA` The Collected_Array object, updated to correspond with the input Distributed_-Vector object.

The Gather_and_Collect_CA_from_DV code listing in § D.8.7 on page 453 contains additional documentation.

9.8.8 Get_Locus_Collected_Array Procedure

The Get_Many_Locus_CA and Get_One_Locus_CA procedures return the loci of the Many and One Structures of the Collected Array respectively.

Calling syntax:

```
Character = Many_Locus(CA)    or
Character = One_Locus(CA)
```

Input variable:

`CA` The Collected_Array object to be queried.

Output variable:

`Locus` The specified locus of the Collected_Array object.

The Get_Locus_Collected_Array code listing in § D.8.8 on page 455 contains additional documentation.

9.8.9 Get_Name_Collected_Array Procedure

The Get_Name_Collected_Array procedure returns the name of the Collected Array.

Calling syntax:

```
Character = Name(CA)
```

Input variable:

`CA` The Collected_Array object to be queried.

Output variable:

`Name` The name of the Collected_Array object.

The Get_Name_Collected_Array code listing in § D.8.9 on page 456 contains additional documentation.

9.8.10 Get_Values_Collected_Array Procedure

The Get_Values_Collected_Array procedure gets the values from an Collected Array, in the form of a Bare Naked Array.

Calling syntax:

```
Values = CA                    or
call Get_Values (Values, CA)
```

Input variable:

`CA` The Collected_Array object to be queried.

Output variable:

`Values` The Bare Naked Array of values from the Collected_Array object, defined differently on each PE.

The Get_Values_Collected_Array code listing in § D.8.10 on page 456 contains additional documentation.

9.8.11 Get_Version_Collected_Array Procedure

The Get_Version_Collected_Array procedure returns the version number for the Collected Array.

Calling syntax:

```
Integer = Version(CA)
```

Input variables:

`CA` The Collected_Array object to be queried.

Output variable:

`Version` The version number of the Collected_Array object.

The Get_Version_Collected_Array code listing in § D.8.11 on page 457 contains additional documentation.

9.8.12 Output_Collected_Array Procedure

The Output_Collected_Array procedure writes out a section of a Collected Array to the specified unit.

Calling syntax:

```
call Output (CA, One_First, One_Last, Unit)
```

Input variables:

`CA`	The Collected_Array object to be queried.
`One_First`	The first location on the One Axis to be output. [Optional]
`One_Last`	The last location on the One Axis to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]

The Output_Collected_Array code listing in § D.8.12 on page 458 contains additional documentation.

9.8.13 Set_Values_Collected_Array Procedure

The Set_Values_Collected_Array procedure sets the values for the Collected Array.

Calling syntax:

```
CA = Values                    or
call Set_Values (CA, Values)
```

Input variable:

`Values` The Bare Naked Array of values for the Collected_Array object, defined differently on each PE.

Input/Output variable:

`CA` The Collected_Array object to be set.

Internal variable:

`Version_Increment` The amount that the version number is incremented, which is a global class variable.

The Set_Values_Collected_Array code listing in § D.8.13 on page 463 contains additional documentation.

9.8.14 Set_Version_Collected_Array Procedure

The Set_Version_Collected_Array procedure sets the version number for the Collected Array.

Calling syntax:

```
CA = Version                    or
call Set_Version (CA, Version)
```

Input variable:

`Version` The version number for the Collected_Array object.

Input/Output variable:

`CA` The Collected_Array object to be set.

The Set_Version_Collected_Array code listing in § D.8.14 on page 464 contains additional documentation.

Chapter 10

Mathematics Module

> *To those who do not know mathematics it is difficult to get across a real feeling as to the beauty, the deepest beauty of nature. If you want to learn about nature, to appreciate nature, it is necessary to understand the language that she speaks in. – Richard Feynman (1918-1988)*
>
> *Anyone who cannot cope with mathematics is not fully human. At best he is a tolerable subhuman who has learned to wear boots, bathe, and not make messes in the house. – Robert Heinlein, "Time Enough for Love"*
>
> *. . . beware of mathematicians and all those who make empty prophecies. The danger already exists that mathematicians have made a covenant with the devil to darken the spirit and confine man in the bonds of Hell. – St. Augustine, DeGenesi ad Litteram*

Many mathematical constructions are repeated over and over in a large computer code. In the CÆSAR code package, they are all grouped together to ensure that they are done in a consistent, correct manner which is easily updated.

The Mathematics Module code listing in § E on page 471 contains additional documentation.

10.1 Math_Utils Module

The Math_Utils Module provides utility routines to solve mathematical problems for the CÆSAR Code Package.

The Math_Utils methods section in § 15.1 on page 187 describes the methods used in the Math_Utils Class.

Math_Utils public procedures:

`Prime_Factors` Returns a vector containing the prime factorization of a number.

The Math_Utils Module code listing in § E.1 on page 471 contains additional documentation. The Math_Utils Module also contains a Unit Test Program which is listed in § E.1.2 on page 475.

10.1.1 Prime_Factors_Math_Utils Procedure

The Prime_Factors_Math_Utils procedure returns a vector of the prime factors for a given number. For negative numbers, it returns a vector with "-1" in the first position and then the prime factorization for the corresponding positive number. For zero, it returns a single zero as the prime factorization. The procedure gives correct answers for all integers in the range [-HUGE(),HUGE()].

Calling syntax:

```
call Prime_Factors (Number, NFactors, Factors, Verbose)
```

Input variables:

`Number`	The number to be factored.
`Verbose`	Toggle for output. Default is false. [optional]

Output variables:

`NFactors`	The number of prime factors.
`Factors(32)`	A vector of the prime factors.

The Prime_Factors_Math_Utils code listing in § E.1.1 on page 472 contains additional documentation.

10.2 Statistics Class

The Statistics Class is used to describe a Statistics object in the CÆSAR Code Package. A Statistics object contains statistical information about a set of values that have been added to it. The statistical information that is available includes means (arithmetic, geometric and harmonic), extrema and the standard deviation. It does not include information that would require the storage of the entire set of values, such as median and mode.

The Statistics methods section in § 15.2 on page 188 describes the methods used in the Statistics Class.

Statistics public procedures:

Fundamental procedures	
`Initialize`	Initializes a Statistics object.
`Finalize`	Finalizes a Statistics object.
`Valid_State`	Returns false iff a Statistics object is in an invalid state.
`Initialized`	Returns true iff a Statistics object has been initialized.
Operations	
`Add_Value`	Adds a value to the data set of the Statistics object.
`Arithmetic_Mean`	Returns the arithmetic mean of the Statistics object.
`Average`	Returns the arithmetic mean of the Statistics object.
`Count`	Returns the number of values in the Statistics object.
`Geometric_Mean`	Returns the geometric mean of the Statistics object.
`Harmonic_Mean`	Returns the harmonic mean of the Statistics object.
`Maximum`	Returns the maximum value of the Statistics object.
`Mean`	Returns the arithmetic mean of the Statistics object.
`Minimum`	Returns the minimum value of the Statistics object.
`Name`	Returns the name of the Statistics object.
`Output`	Writes out the Statistics object.
`Standard_Deviation`	Returns the standard deviation of the Statistics object.
`Sum`	Returns the sum or total of the Statistics object.
`Total`	Returns the sum or total of the Statistics object.
`Totally_Positive`	Returns true if all the values in the Statistics object are positive.
`Update_Global`	Communicates with all processors to update the global information for the Statistics object.

Statistics public defined types:

Statistics type	
`Global_Arithmetic_Mean`	The global arithmetic mean.
`Global_Count`	The global number of items in the data set.
`Global_Geometric_Mean`	The global geometric mean.
`Global_Harmonic_Mean`	The global harmonic mean.
`Global_Log_Sum`	The global sum of the log of the values.
`Global_Maximum`	The global maximum of the values.
`Global_Minimum`	The global minimum of the values.
`Global_Reciprocal_Sum`	The global sum of the reciprocal of the values.
`Global_Squared_Sum`	The global sum of the squares of the values.
`Global_Standard_Deviation`	The global standard deviation.
`Global_Sum`	The global sum of values.
`Global_Totally_Positive`	True if all items in the data set on all PEs are positive.
`Global_Updated`	Global update status.
`Initialized`	Initialization status.
`Name`	The name for this Statistics object.
`PE_Arithmetic_Mean`	The arithmetic mean on this PE.
`PE_Count`	The number of items in the data set on this PE.
`PE_Geometric_Mean`	The geometric mean on this PE.
`PE_Harmonic_Mean`	The harmonic mean on this PE.
`PE_Log_Sum`	The sum of the log of the values on this PE.
`PE_Maximum`	The maximum of the values on this PE.
`PE_Minimum`	The minimum of the values on this PE.
`PE_Reciprocal_Sum`	The sum of the reciprocal of the values on this PE.
`PE_Squared_Sum`	The sum of the squares of the values on this PE.
`PE_Standard_Deviation`	The standard deviation on this PE.
`PE_Sum`	The sum of values on this PE.
`PE_Totally_Positive`	True if all items in the data set on this PE are positive.

The Statistics Class code listing in § E.2 on page 476 contains additional documentation. The Statistics Class also contains a Unit Test Program which is listed in § E.2.9 on page 493.

10.2.1 Initialize_Statistics Procedure

The Initialize_Statistics procedure allocates and initializes a Statistics object.

Calling syntax:

```
call Initialize (Statistics, Name, status)
```

Input variables:

`Statistics`	The Statistics object to be initialized.
`Name`	The name for the Statistics object.

Output variables:

`Statistics`	The Statistics object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Statistics code listing in § E.2.1 on page 478 contains additional documentation.

10.2.2 Finalize_Statistics Procedure

The Finalize_Statistics procedure deallocates and finalizes a Statistics object.

Calling syntax:

```
call Finalize (Statistics, status)
```

Input variables:

`Statistics`	The Statistics object to be finalized.

Output variables:

`Statistics`	The Statistics object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Statistics code listing in § E.2.2 on page 481 contains additional documentation.

10.2.3 Valid_State_Statistics Procedure

The Valid_State_Statistics procedure returns true iff the Statistics is in a valid state – that is, iff the Statistics passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Statistics)
```

Input variables:

`Statistics`	The Statistics to be checked.

Output variable:

`Valid_State`	True iff the Statistics is in a valid state.

Internal variables:

`Global_Mean_Range`	Global range of the mean.

`Global_Range`	Global range of the distribution.
`PE_Mean_Range`	Range of the mean on this PE.
`PE_Range`	Range of the distribution on this PE.

The Valid_State_Statistics code listing in § E.2.3 on page 482 contains additional documentation.

10.2.4 Initialized_Statistics Procedure

The Initialized_Statistics procedure returns true iff the Statistics object has been initialized.

Calling syntax:

```
Logical = Initialized(Statistics)
```

Input variable:

`Statistics`	The Statistics object to be examined.

Output variable:

`Initialized`	True iff the Statistics object has been initialized.

The Initialized_Statistics code listing in § E.2.4 on page 485 contains additional documentation.

10.2.5 Add_Value_Statistics Procedure

The Add_Value_Statistics procedure adds a new value to the Statistics object.

Calling syntax:

```
call Add_Value (Statistics, Value)
```

Input variables:

`Statistics`	The Statistics object to be modified.
`Value`	Value to be added to Statistics object.

Output variables:

`Statistics`	The modified Statistics object.

Internal variables:

`N`	Real version of PE_Count.

The Add_Value_Statistics code listing in § E.2.5 on page 485 contains additional documentation.

10.2.6 Get Value Statistics Functions

The Get Value Statistics functions return values from a Statistics object. Local or global values can be returned by specifying the Global logical input variable.

Calling syntax:

```
Output = Arithmetic_Mean (Statistics, Global)      ,
Output = Average (Statistics, Global)              ,
Output = Count (Statistics, Global)                ,
Output = Geometric_Mean (Statistics, Global)       ,
Output = Harmonic_Mean (Statistics, Global)        ,
Output = Maximum (Statistics, Global)              ,
Output = Mean (Statistics, Global)                 ,
Output = Minimum (Statistics, Global)              ,
Output = Name (Statistics)                         ,
Output = Standard_Deviation (Statistics, Global)   ,
Output = Sum (Statistics, Global)                  ,
Output = Total (Statistics, Global)                or
Output = Totally_Positive (Statistics, Global)
```

Mean and Average are alternate interface names for the Arithmetic_Mean Procedure, and Total is an alternate interface name for the Sum Procedure.

Input variables:

`Statistics`	The Statistics object to be examined.
`Global`	If present and true, the function does a global update and then returns the global value. Otherwise, the local PE value is returned. [Optional]

Output variable:

`Output`	For Name, returns a character variable containing the name assigned to the object upon initialization. For Count, returns the integer number of values in the Statistics object. For Totally_Positive, returns a logical which is true only if all the values are positive. For all other functions, returns a real variable with the named value for the Statistics object. Note that the values returned by Geometric_Mean and Harmonic_Mean have no significance if Totally_Positive is false.

The Get Value Statistics code listing in § E.2.6 on page 487 contains additional documentation.

10.2.7 Output_Statistics Procedure

The Output_Statistics procedure writes out information from a Statistics object to the specified unit.

Calling syntax:

```
call Output (Statistics, Global, Verbose, Unit)
```

Input variables:

`Statistics`	The Statistics object to be queried.
`Global`	Global flag, defaults to false. If Global is not set to true, no global update is done and whatever value is present on the IO_PE is used. [Optional]
`Verbose`	Verbosity flag, defaults to false. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]

The Output_Statistics code listing in § E.2.7 on page 489 contains additional documentation.

10.2.8 Update_Global_Statistics Procedure

The Update_Global_Statistics procedure does interprocessor communication to update the global values of a Statistics object.

Calling syntax:

```
call Update_Global (Statistics)
```

Input variables:

`Statistics`	The Statistics object to be globally updated.

Output variables:

`Statistics`	The globally updated Statistics object.

Internal variables:

`N`	Real version of Global_Count.

The Update_Global_Statistics code listing in § E.2.8 on page 492 contains additional documentation.

Chapter 11

Parallel_Utilities Module

> *His narrative style was like parallel parking on a busy street: he proceeded in fits and starts, backed up, edged forward, backed up, while other ideas zoomed past close at hand. – Melissa Fay Greene*

The Parallel Utilities module provides CÆSAR with utilities of a general nature which rely on parallel communication for their operation.

The Parallel_Utilities Module code listing in § F on page 497 contains additional documentation.

11.1 Timer Class

The Timer Class is used to keep track of run times in the CÆSAR Code Package. A Timer keeps track of both CPU and Wall Clock times. The time between starting and stopping a Timer is referred to as a split time. A Timer keeps statistics for split and total times, and for each processor in a parallel run.

Timer public procedures:

Fundamental procedures	
`Initialize`	Initializes a Timer object.
`Finalize`	Finalizes a Timer object.
`Valid_State`	Returns false iff a Timer object is in an invalid state.
Operations	
`Arithmetic_Mean`	Returns the arithmetic mean of the Timer object.
`Average`	Returns the arithmetic mean of the Timer object.
`Count`	Returns the number of splits in the Timer object.
`Geometric_Mean`	Returns the geometric mean of the Timer object.
`Get_CPU_Time`	Returns CPU time in seconds.
`Get_Wall_Clock_Time`	Returns Wall Clock time in seconds.
`Harmonic_Mean`	Returns the harmonic mean of the Timer object.
`Initialized`	Returns true iff a Timer object has been initialized.
`Maximum`	Returns the maximum value of the Timer object.
`Mean`	Returns the arithmetic mean of the Timer object.
`Minimum`	Returns the minimum value of the Timer object.
`Julian_Day`	Returns the Julian Day number for a given date.
`Name`	Returns the name of the Timer object.
`Output`	Writes out the Timer object.
`Reset`	Resets all the registers in a Timer object, returning it to a freshly initialized state.

`Standard_Deviation`	Returns the standard deviation of the Timer object.
`Start`	Instructs a Timer object to begin timing.
`Stop`	Instructs a Timer object to discontinue timing.
`Sum`	Returns the sum or total of the Timer object.
`Total`	Returns the sum or total of the Timer object.
`Totally_Positive`	Returns true if all the times in the Timer object are positive.

Timer public defined types:

Timer type	
`CPU_Time`	Time object to track.
`Initialized`	Initialization status.
`Name`	The name for this Timer object.
`Running`	Logical for Timer state.
`Wall_Clock_Time`	Time object to track.
Time type	
`Start`	Initial time value.
`Statistics`	Statistics for the time splits.

The Timer Class code listing in § F.1 on page 497 contains additional documentation. The Timer Class also contains a Unit Test Program which is listed in § F.1.13 on page 519.

11.1.1 Initialize_Timer Procedure

The Initialize_Timer procedure allocates and initializes a Timer object.

Calling syntax:

```
call Initialize (Timer, Name, status)
```

Input variables:

`Timer`	The Timer object to be initialized.
`Name`	The name for the Timer object.

Output variables:

`Timer`	The Timer object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Timer code listing in § F.1.1 on page 500 contains additional documentation.

11.1.2 Finalize_Timer Procedure

The Finalize_Timer procedure deallocates and finalizes a Timer object.

Calling syntax:

```
call Finalize (Timer, status)
```

Input variables:

`Timer`	The Timer object to be finalized.

Output variables:

`Timer`	The Timer object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Timer code listing in § F.1.2 on page 502 contains additional documentation.

11.1.3 Valid_State_Timer Procedure

The Valid_State_Timer procedure returns true iff the Timer is in a valid state – that is, iff the Timer passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Timer)
```

Input variables:

`Timer`	The Timer to be checked.

Output variable:

`Valid_State`	True iff the Timer is in a valid state.

Internal variables:

`Variable`	Description. [Units]

The Valid_State_Timer code listing in § F.1.3 on page 503 contains additional documentation.

11.1.4 Initialized_Timer Procedure

The Initialized_Timer procedure returns true iff the Timer object has been initialized.

Calling syntax:

```
Logical = Initialized(Timer)
```

Input variable:

`Timer`	The Timer object to be examined.

Output variable:

`Initialized`	True iff the Timer object has been initialized.

The Initialized_Timer code listing in § F.1.4 on page 504 contains additional documentation.

11.1.5 Get Value Timer Functions

The Get Value Timer functions return values from a Timer object. Local or global values can be returned by specifying the Global logical input variable. Note that if the Timer object is currently running, data from the current timing is not included. Therefore, it is more meaningful to call these routines after calling Stop_Timer.

Calling syntax:

```
Output = Arithmetic_Mean (Timer, Clock, Global, Split)      ,
Output = Average (Timer, Clock, Global, Split)              ,
Output = Count (Timer, Clock, Global, Split)                ,
Output = Geometric_Mean (Timer, Clock, Global, Split)       ,
Output = Harmonic_Mean (Timer, Clock, Global, Split)        ,
Output = Maximum (Timer, Clock, Global, Split)              ,
Output = Mean (Timer, Clock, Global, Split)                 ,
Output = Minimum (Timer, Clock, Global, Split)              ,
Output = Name (Timer)                                       ,
Output = Standard_Deviation (Timer, Clock, Global, Split)   ,
Output = Sum (Timer, Clock, Global, Split)                  ,
Output = Total (Timer, Clock, Global, Split)                or
Output = Totally_Positive (Timer, Clock, Global, Split)
```

Mean and Average are alternate interface names for the Arithmetic_Mean Procedure, and Total is an alternate interface name for the Sum Procedure.

Note that some combinations of options will return values that, while correct, have little useful meaning. For example, the function `Total(Timer, Clock=''Wall_Clock'', Global=.true., Split=.false.)` will return the total wall-clock time summed over all the processors, whereas a more meaningful value would be the maximum wall-clock time over all the processors, given by `Maximum(Timer, Clock=''Wall_Clock'', Global=.true., Split=.false.)` .

Input variables:

`Timer`	The Timer object to be examined.
`Clock`	Should have a value of "CPU" or "Wall_Clock". Default is "CPU". [Optional]
`Global`	If present and true, the function does a global update and then returns the global value. Otherwise, the local PE value is returned. [Optional]

`Split` If present and true, the function returns values based on the individual split times, that is, for all the times that the timer was started and stopped. This can be either local (all the splits on a PE) or global (all the splits on all the PEs). If Split is false or absent, the function returns values based on the total times on each PE. If both Split and Global are false, the function returns values based on the single entry of the total time on the local PE. [Optional]

Output variables:

`Output` For Name, returns a character variable containing the name assigned to the object upon initialization. For Count, returns the integer number of values in the Timer object (i.e. the number of splits). For Totally_Positive, returns a logical which is true only if all the values are positive. For all other functions, returns a real variable with the named value for the Timer object. Note that the values returned by Geometric_Mean and Harmonic_Mean have no significance if Totally_Positive is false.

The Get Value Timer code listing in § F.1.5 on page 505 contains additional documentation.

11.1.6 Get_CPU_Time Procedure

The Get_CPU_Time procedure returns the current value of the CPU time counter in seconds. It is intended to be used internally to the Timer Class, but is made public in case it is needed elsewhere.

Note: on an old Sun compiler (Sun WorkShop 6 update 2 Fortran 95 6.2 2001/05/15), the CPU_TIME call returns the total time for the parent process, so interpret results carefully. Some runs led me to suspect that CPU_TIME returns Wall Clock Time on Suns – not sure about this.

Calling syntax:

```
Real = Get_CPU_Time ()
```

Output variable:

`Get_CPU_Time` The current value of the CPU time counter [seconds].

The Get_CPU_Time code listing in § F.1.6 on page 507 contains additional documentation.

11.1.7 Get_Wall_Clock_Time Procedure

The Get_Wall_Clock_Time procedure returns the current value of the Wall Clock time counter in seconds. It is intended to be used internally to the Timer Class, but is made public in case it is needed elsewhere.

The returned value is actually the number of seconds, down to the millisecond, since the midnight at the beginning of 4714 BC/11/24 (-4713 CE/11/24). This is the beginning date of the Julian Day counter according to the Gregorian calendar, which is 38 days before the beginning date of the Julian Day counter according to the Julian calendar (4713 BC/01/01 or -4712 CE/01/01).

Calling syntax:

```
Real = Get_Wall_Clock_Time ()
```

Output variable:

`Get_Wall_Clock_Time` The current value of the Wall Clock time counter [seconds].

The Get_Wall_Clock_Time code listing in § F.1.7 on page 508 contains additional documentation.

11.1.8 Julian_Day Procedure

The Julian_Day procedure calculates the Julian Day for a given Year, Month and Day for a specific calendar (Julian or Gregorian).

Days in a Year

The Romans originally counted years *ab urbe condita* (a.u.c.), that is, "from the founding of the city (Rome)", posited as April 21, 753 BC. However, common usage numbered years by starting at one every time a new Caesar's reign began. Days were added and subtracted from time to time to keep the calendar on track with the season, which led to problems with planning and abuses for political purposes. (To calculate the equivalent a.u.c. year, add 753 to an AD year or subtract a BC year from 754. This is a rough estimate and may be off by a year.)

To correct problems with this calendar, Julius Caesar instituted a new calendar in 709 a.u.c. (45 BC). This calendar, called the Julian calendar, had years of 365 days and leap years of 366 days every fourth year. Months had a constant number of days, with a new month being added to refer to Julius Caesar himself (July). The year of 45 BC is often called the "year of confusion", as 90 days were added to the year to realign with the seasons.

The Julian calendar was a distinct improvement, but its year length of 365.25 days differed from the true year by about 11 minutes. Two possible values for the true year length are the vernal equinox year, 365.2424 days, and the tropical year, 365.24219 days. The discrepancy in the Julian calendar accumulates to a day roughly every 131 years.

Over the centuries the equinoxes and solstices migrated from their former places in the calendar, which caused problems in the Roman Catholic Church because the determination of the date of Easter was affected. Eventually, Pope Gregory XIII issued a papal bull establishing what is now called the Gregorian calendar. The Gregorian calendar modified the addition of leap days, such that a century year (divisible by 100) was only counted as a leap year if it was also divisible by 400. The Gregorian year length of 365.2425 days is much closer to the true year length, differing by somewhere between 8 and 27 seconds (depending on which true year is used). An entire day of error accumulates in either 3,226 or 10,000 years.

Adoption of the Gregorian calendar required dropping some days to realign the equinoxes and solstices. The first adoption of the Gregorian calendar was by the Catholic countries of Italy, Poland, Portugal, and Spain. They dropped 10 days, following 4 October 1582 immediately with 15 October 1582. Other Catholic countries followed suit, Protestant countries changed over the next two hundred years, and Greek Orthodox countries (Russia, Romania, Bulgaria, Turkey) did not change until the early 1900s. Great Britain and colonies (including American colonies) dropped 11 days when they adopted the Gregorian calendar, following 2 September 1752 with 14 September 1752. (Unix users can see that this fact is incorporated into the cal program by typing "`% cal 9 1752`".)

Extending a calendar before the time when it was adopted is referred to as the "proleptic" version of that calendar.

Year Numbering Systems

The designation "A.D." is an abbreviation of "Anni Domini Nostri Jesu Christi", i.e., "in the year of Our Lord Jesus Christ", and "B.C." signifies "Before Christ". There was no year zero in this system – that is, the year 1 BC was followed by 1 AD. A more religiously-neutral system replaces A.D. with C.E. (for Christian Era) and B.C. with B.C.E. (Before Christian Era), and leaves the numbering unchanged.

A third system is called the Common Era calendar, which uses the designation C.E. only. It includes a zero year and negative years, so that 2000 AD = 2000 CE (Christian Era) = 2000 CE (Common Era), and 2 BC = 2 BCE (Before Christian Era) = -1 CE (Common Era).

Julian Days

Table 11.18: Chronological Julian Day numbers for some representative dates.

Date (BC/AD)	Date (CE)	Julian Day (Julian Calendar)	Julian Day (Gregorian Calendar)
4714 BC/11/24	-4713 CE/11/24	-38	0
4713 BC/01/01	-4712 CE/01/01	0	38
753 BC/04/21	-752 CE/04/21	1446501	1446509
2 BC/10/30	-1 CE/10/30	1720995	1720997
1 BC/01/01	0 CE/01/01	1721058	1721060
1 AD/01/01	1 CE/01/01	1721424	1721426
200 AD/02/28	200 CE/02/28	1794166	1794167
200 AD/02/29	200 CE/02/29	1794167	1794168
200 AD/03/01	200 CE/03/01	1794168	1794168
300 AD/02/28	300 CE/02/28	1830691	1830691
300 AD/02/29	300 CE/02/29	1830692	1830692
300 AD/03/01	300 CE/03/01	1830693	1830692
1582 AD/10/04	1582 CE/10/04	2299160	2299150
1582 AD/10/14	1582 CE/10/14	2299170	2299160
1752 AD/09/02	1752 CE/09/02	2361221	2361210
1752 AD/09/13	1752 CE/09/13	2361232	2361221
1858 AD/11/16	1858 CE/11/16	2400012	2400000
1968 AD/05/23	1968 CE/05/23	2440013	2440000
1995 AD/10/09	1995 CE/10/09	2450013	2450000
2000 AD/01/01	2000 CE/01/01	2451558	2451545
2132 AD/08/31	2132 CE/08/31	2500014	2500000

Julian Day numbers had their beginnings in a numbering system that was designed by Joseph Scaliger in 1583. It is sometimes erroneously stated that the system was named to honor his father, Julius Caesar Scaliger, but Scaliger himself wrote "We have termed it Julian because it fits the Julian year ...". His system was based on a 7980-year cycle which started on 1 January 4713 BC in the proleptic Julian calendar.

The astronomer John W. F. Herschel extended this idea to a system which numbered all the days consecutively starting at noon on 1 January -4712 CE (Julian). Noon was used so that the day number would not change in the middle of a night observation. Chronological Julian Day numbers were subsequently defined to start at midnight at the start of 1 January -4712 CE (Julian).

Some representative Julian Day numbers are given in Table 11.1.8.

See `http://www.hermetic.ch/cal_stud/jdn.htm`, `http://www.hermetic.ch/cal_stud/cal_art.htm`, `http://penelope.uchicago.edu/~grout/encyclopaedia_romana/calendar/consuls.html`, `http://en.wikipedia.org/wiki/Ab_urbe_condita`, `http://en.wikipedia.org/wiki/Julian_calendar`, `http://en.wikipedia.org/wiki/Gregorian_calendar`, `http://en.wikipedia.org/wiki/Julian_day`, and `http://astro.nmsu.edu/~lhuber/leaphist.html` for more detailed discussions.

Calling syntax:

```
integer = Julian_Day(Year, Month, Day, Calendar, Debug)
```

Input variables:

`Year`	Year number, with negative numbers for BC and positive numbers for AD. There is no year zero. This procedure will give accurate results (corresponding to the proleptic Julian and Gregorian calendars) from 4713 BC onward. [years]
`Month`	Month number. [1–12]
`Day`	Day number. [1–31]
`Calendar`	Character variable specifying whether to use the Julian or Gregorian calendar (Gregorian is the default). [optional]

`Debug`	Debug toggle, needed to turn off very slow checking during 1000s of repeated calls in the parallel versions of the unit test. (.true. is the default). [optional]

Output variables:

`Julian_Day`	The chronological Julian Day number for the input date. [days]

Internal variables:

`Julian_Day_Constant`	The number of days between the zero date of the Julian Day numbering system (1 January -4712 CE) and the zero date of this Julian Day calculation (30 October -1 CE). [days]
`Julian_Month`	A month number which has been adjusted for ease of calculation. Ranges between 4 and 15.
`Julian_Year`	The year, with negative values shifted by one so that there is a "zero year" (there was no zero year between 1 BC and 1 AD). This is also known as the Common Era (CE) year. [years]
`Shifted_Julian_Year`	The Julian year shifted by 8000 years for ease of calculation of leap days. [years]

The Julian_Day code listing in § F.1.8 on page 509 contains additional documentation.

11.1.9 Output_Timer Procedure

The Output_Timer procedure writes out information from a Timer object to the specified unit.

Calling syntax:

```
call Output (Timer, Global, Verbose, Unit)
```

Input variables:

`Timer`	The Timer object to be queried.
`Global`	Global flag, defaults to false. If Global is not set to true, no global update is done and whatever value is present on the IO_PE is used. [Optional]
`Verbose`	Verbosity flag, defaults to false. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]

The Output_Timer code listing in § F.1.9 on page 512 contains additional documentation.

11.1.10 Reset_Timer Procedure

The Reset_Timer procedure resets the Timer registers. A Timer that has been reset is identical to one that has just been initialized.

Calling syntax:

```
call Reset_Timer (Timer)
```

Input variables:

`Timer`	The Timer object to be reset.

Output variables:

`Timer` The Timer object that has been reset.

The Reset_Timer code listing in § F.1.10 on page 517 contains additional documentation.

11.1.11 Start_Timer Procedure

The Start_Timer procedure starts the Timer running. The Timer registers are not reset to zero.

Calling syntax:

```
call Start_Timer (Timer)
```

Input variables:

`Timer` The Timer object to be started.

Output variables:

`Timer` The Timer object that is now running.

The Start_Timer code listing in § F.1.11 on page 517 contains additional documentation.

11.1.12 Stop_Timer Procedure

The Stop_Timer procedure stops the Timer. The Timer registers are not reset to zero.

Calling syntax:

```
call Stop_Timer (Timer)
```

Input variables:

`Timer` The Timer object to be stopped.

Output variables:

`Timer` The Timer object that has been stopped.

The Stop_Timer code listing in § F.1.12 on page 518 contains additional documentation.

Chapter 12

Linear_Algebra Module

That fondness for science, ... has encouraged me to compose a short work on calculating by al-jabr and al-muqabala, confining it to what is easiest and most useful in arithmetic, such as men constantly require in cases of inheritance, legacies, partition, lawsuits, and trade, and in all their dealings with one another, or where the measuring of lands, the digging of canals, geometrical computations, and other objects of various sorts and kinds are concerned. – from the algebra treatise Hisab al-jabr w'al-muqabala, the most famous work of Abu Ja'far Muhammad ibn Musa Al-Khwarizmi (c. 780 – c. 850) [al-jabr means "restoring", referring to the process of moving a subtracted quantity to the other side of an equation; al-muqabala is "comparing" and refers to subtracting equal quantities from both sides of an equation.]

The Linear_Algebra Module contains classes to support the manipulation and solution of linear algebra problems for the CÆSAR Code Package. Several matrix classes (with different storage formats) and a Mathematic Vector class are included. A Solver class is used to drive the solution of linear equations using both external packages and solvers included within CÆSAR.

The Linear_Algebra Module code listing in § G on page 531 contains additional documentation.

12.1 Mathematic_Vector Class

The Mathematic_Vector Class is used to describe a Mathematic Vector in the CÆSAR Code Package. A Mathematic_Vector differs from a Distributed_Vector in several ways:

- A Mathematic_Vector has operations which allow it to calculate various mathematic functions (norms, averages, etc.).
- A Mathematic_Vector is used with a matrix class to define a matvec operation.
- A Distributed_Vector can interoperate with other CÆSAR parallel data structures (Assembled_Vector, Overlapped_Vector, Collected_Array).

Note that a Mathematic_Vector contains a Distributed_Vector and a vector of Overlapped_Vectors which it uses for matvecs.

The Mathematic_Vector methods section in § 16.1 on page 189 describes the methods used in the Mathematic_Vector Class.

Mathematic_Vector public procedures:

Fundamental procedures	
`Initialize`	Initializes a Mathematic_Vector object.
`Finalize`	Finalizes a Mathematic_Vector object.
`Valid_State`	Returns false iff a Mathematic_Vector object is in an invalid state.
`Initialized`	Returns true iff a Mathematic_Vector object has been initialized.
Operations	
`Average`	Returns the arithmetic mean of the Mathematic_Vector object.
`Add_Values`	Increments the values of a Mathematic_Vector object.
`DotProduct`	Returns the global dot product of two Mathematic_Vector objects.
`Duplicate`	Makes an exact copy of a Mathematic_Vector (except for the internal DV's and OV's, which will be generated if needed). This procedure is also useful when a compatible Mathematic_Vector is needed.
`First_PE`	Returns the global number of the first entry on this PE of the Mathematic_Vector object.
`Get_Values`	Returns the values for a Mathematic_Vector object (also has an assignment interface).
`Infinity_Norm`	Returns the infinity norm of the Mathematic_Vector object.
`Last_PE`	Returns the global number of the last entry on this PE of the Mathematic_Vector object.
`Length_PE`	Returns the length on this PE of the Mathematic_Vector object.
`Length_Total`	Returns the total length (all PEs) of the Mathematic_Vector object.
`Locus`	Returns the locus of the Mathematic_Vector object.
`Maximum`	Returns the maximum of the Mathematic_Vector object.
`Mean`	Returns the mean value of the Mathematic_Vector object.
`Minimum`	Returns the minimum of the Mathematic_Vector object.
`Name`	Returns the name of the Mathematic_Vector object.
`Norm`	Returns the two norm of the Mathematic_Vector object.
`One_Norm`	Returns the one norm of the Mathematic_Vector object.
`Orthogonal`	Returns true if the two Mathematic_Vector objects are orthogonal.
`Output`	Writes out the Mathematic_Vector object.
`P_Norm`	Returns the P-norm of the Mathematic_Vector object.
`Two_Norm`	Returns the two norm of the Mathematic_Vector object.
`Set_Not_Up_to_Date`	Puts a Mathematic Vector into a "not up-to-date" state. Can be used externally to force recalculation of norms, extrema, etc.
`Set_Value`	Sets all values to a specified scalar (often zero) for a Mathematic_Vector object.
`Set_Values`	Sets the values for a Mathematic_Vector object.
`Sum`	Returns the sum of the Mathematic_Vector object.
`Total`	Returns the sum of the Mathematic_Vector object.
`Update_DV`	Updates the Distributed Vector inside of a Mathematic_Vector object.

Mathematic_Vector public defined types:

Mathematic_Vector type	
`Average`	Global average of the Mathematic Vector.
`Average_is_Updated`	Updated? toggle for Average.
`Dimensionality`	Number of dimensions for the Mathematic Vector (always unity).
`DV`	Distributed Vector that is used for matvecs.
`Infinity_Norm`	Infinity norm of the Mathematic Vector.
`Infinity_Norm_is_Updated`	Updated? toggle for Infinity_Norm.
`Initialized`	Initialization status.
`Maximum`	Global maximum of the Mathematic Vector.
`Maximum_is_Updated`	Updated? toggle for Maximum.
`Minimum`	Global minimum of the Mathematic Vector.
`Minimum_is_Updated`	Updated? toggle for Minimum.

`Name`	The name for this variable (especially useful in a vector of Mathematic Vectors).
`One_Norm`	One norm of the Mathematic Vector.
`One_Norm_is_Updated`	Updated? toggle for One_Norm.
`OV`	A vector of Overlapped Vectors that is used for matvecs.
`DV_is_Updated`	Updated? toggle for DV.
`P_Norm`	P norm of the Mathematic Vector.
`P_Norm_Exponent`	Exponent used in taking the P norm.
`P_Norm_is_Updated`	Updated? toggle for P_Norm.
`Structure`	Basic data structure for the Mathematic Vector.
`Sum`	Global sum of the Mathematic Vector.
`Sum_is_Updated`	Updated? toggle for Sum.
`Two_Norm`	Two norm of the Mathematic Vector.
`Two_Norm_is_Updated`	Updated? toggle for Two_Norm.
`Values`	Values for the Mathematic Vector.

Mathematic_Vector public variables:

`Number_of_OVs_in_an_MV`	A parameter specifying the number of Overlapped_Vector objects in a Mathematic_-Vector object. This number limits the number of different matrices that MatVecs can be done with quickly. For example, if a Mathematic_Vector object is to be multipied by three matrices, all with the same Column_Structure but with different Data_Indices (Columns arrays), then the set up for all three can be stored as long as `Number_of_OVs_in_an_MV` is equal to or greater than three. Otherwise, multiplication can still be done, but set up phases will have to be repeated.

The Mathematic_Vector Class code listing in § G.1 on page 531 contains additional documentation. The Mathematic_Vector Class also contains a Unit Test Program which is listed in § G.1.15 on page 562.

12.1.1 Initialize_Mathematic_Vector Procedure

The Initialize_Mathematic_Vector procedure allocates and initializes a Mathematic_Vector object.

Calling syntax:

```
call Initialize (MV, Structure, Name, status)
```

Input variables:

`MV`	The Mathematic_Vector object to be initialized.
`Structure`	The Base_Structure giving the distribution for the Mathematic_Vector.
`Name`	The name for this variable (especially useful in a vector of Mathematic Vectors). [Optional]

Output variables:

`MV`	The Mathematic_Vector object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.

`consolidated_status`	Consolidated Status.

The Initialize_Mathematic_Vector code listing in § G.1.1 on page 536 contains additional documentation.

12.1.2 Duplicate_Mathematic_Vector Procedure

The Duplicate_Mathematic_Vector procedure duplicates a Mathematic Vector – that is, it initializes a new Mathematic Vector with the same structure as the old Mathematic Vector, and copies the internals. Note that the internal structures associated with executing MatVecs (i.e. the DV and the OVs) are not duplicated.

Calling syntax:

```
call Duplicate (MV_duplicate, MV_source, status)
```

Input variables:

`MV_source`	The Mathematic_Vector to be duplicated.

Output variables:

`MV_duplicate`	The Mathematic_Vector object is now a copy of the MV_source Mathematic Vector.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

The Duplicate_Mathematic_Vector code listing in § G.1.2 on page 538 contains additional documentation.

12.1.3 Finalize_Mathematic_Vector Procedure

The Finalize_Mathematic_Vector procedure deallocates and finalizes a Mathematic_Vector object.

Calling syntax:

```
call Finalize (MV, status)
```

Input variables:

`MV`	The Mathematic_Vector object to be finalized.

Output variables:

`MV`	The Mathematic_Vector object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status mathematic_vector.
`i`	Loop variable.

The Finalize_Mathematic_Vector code listing in § G.1.3 on page 539 contains additional documentation.

12.1.4 Valid_State_Mathematic_Vector Procedure

The Valid_State_Mathematic_Vector procedure returns true iff the Mathematic_Vector is in a valid state – that is, iff the Mathematic_Vector passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(MV)
```

Input variables:

`MV`	The Mathematic_Vector to be checked.

Output variable:

`Valid_State`	True iff the Mathematic_Vector is in a valid state.

The Valid_State_Mathematic_Vector code listing in § G.1.4 on page 541 contains additional documentation.

12.1.5 Initialized_Mathematic_Vector Procedure

The Initialized_Mathematic_Vector procedure returns true iff the Mathematic_Vector object has been initialized.

Calling syntax:

```
Logical = Initialized(MV)
```

Input variable:

`MV`	The Mathematic_Vector object to be examined.

Output variable:

`Initialized`	True iff the Mathematic_Vector object has been initialized.

The Initialized_Mathematic_Vector code listing in § G.1.5 on page 543 contains additional documentation.

12.1.6 Add_Values_Mathematic_Vector Procedure

The Add_Values_Mathematic_Vector procedure increments the values for the Mathematic Vector by the specified vector of value. Note that if the single value form of this procedure is used, then the same number of procedure calls must take place on every processor – this is required by internal verification calls that use global communication. If a different number of values are to be added on each processor, then the vector form should be used.

Calling syntax:

```
call Add_Values (MV, Value, Row, Global)     ,
call Add_Values (MV, Values, Rows, Global)   or
call Add_Values (MV, Values)
```

Input variable:

`Values`	A vector of values to be added to the Mathematic_Vector object. If the Rows variable is not present, Values must be the same size as the Mathematic_Vector. Otherwise, Values must have a size smaller than or equal to the size of the Mathematic_Vector.
`Rows`	An integer vector of rows for the Values vector. [optional]
`Global`	A toggle between using global or local (on this PE) indices for the Rows variable. Default is true (use global). [optional]
`Value`	A single value to be added to the Mathematic_Vector object. The Row variable must be present for this form of the procedure.
`Row`	The row index for the Value variable.

Input/Output variable:

`MV`	The Mathematic_Vector object to be incremented.

The Add_Values_Mathematic_Vector code listing in § G.1.6 on page 544 contains additional documentation.

12.1.7 DotProduct_Mathematic_Vector Procedure

The DotProduct procedure calculates the global dot product of two Mathematic Vectors.

Note that I would rather call this routine "Dot_Product", but the F90 standard does not allow underscores in defined operator names (i.e. ".Dot_Product." is not allowed).

Calling syntax:

`Output = MV1 .DotProduct. MV2` or
`Output = DotProduct(MV1, MV2)`

Input variables:

`MV1, MV2`	Two Mathematic_Vector objects to be dotted.

Output variable:

`DotProduct`	Scalar result of taking the global dot product of the two Mathematic Vectors.

The DotProduct_Mathematic_Vector code listing in § G.1.7 on page 547 contains additional documentation.

12.1.8 Get Value Mathematic_Vector Functions

The Get_Value_MV functions return values from a Mathematic_Vector object. These operations require global communication (except Get_First_MV, Get_Name_MV, Get_Last_MV, Get_Length_PE and Get_Length_Total), but if called more than once without modifying the object, no global communication is done for the second call.

Calling syntax:

```
Output = Average (MV)          ,
Output = First_PE (MV)         ,
Output = Infinity_Norm (MV)    ,
Output = Length_PE (MV)        ,
Output = Last_PE (MV)          ,
Output = Length_Total (MV)     ,
Output = Locus (MV)            ,
Output = Maximum (MV)          ,
Output = Mean (MV)             ,
Output = Minimum (MV)          ,
Output = Name (MV)             ,
Output = Norm (MV)             ,
Output = One_Norm (MV)         ,
Output = P_Norm (MV, P)        ,
Output = Sum (MV)              ,
Output = Total (MV)            or
Output = Two_Norm (MV)
```

Mean is an alternate interface name for the Average Procedure, Norm is an alternate interface name for the Two_Norm Procedure, and Total is an alternate interface name for the Sum Procedure.

Input variables:

`MV`	The Mathematic_Vector object to be examined.
`P`	P_Norm exponent. This optional argument is only available in P_Norm.

Output variable:

`Output`	For Name and Locus, returns a character variable containing the name or locus assigned to the object upon initialization. For First_PE, Last_PE, Length_PE and Length_Total, returns an integer variable containing the named value of the Mathematic_Vector object. For all other functions, returns a real variable with the named value for the Mathematic_Vector object.

The Get Value Mathematic_Vector code listing in § G.1.8 on page 548 contains additional documentation.

12.1.9 Get_Values_Mathematic_Vector Procedure

The Get_Values_Mathematic_Vector procedure accesses the values from a Mathematic Vector.

Calling syntax:

```
Values = MV                        or
call Get_Values (Values, MV)
```

Input variable:

`Values`	The bare naked vector of values from the Mathematic_Vector object. Values must be the same size as the Mathematic_Vector.

Input/Output variable:

`MV`	The Mathematic_Vector object to be accessed.

The Get_Values_Mathematic_Vector code listing in § G.1.9 on page 552 contains additional documentation.

12.1.10 Orthogonal_Mathematic_Vector Procedure

The Orthogonal procedure calculates whether or not two Mathematic Vectors are orthogonal.

Calling syntax:

```
Logical = MV1 .Orthogonal.  MV2    or
Logical = Orthogonal(MV1, MV2)
```

Input variables:

`MV1, MV2`	Two Mathematic_Vector objects to be dotted.

Output variable:

`Orthogonal`	Logical which is true when the two Mathematic Vectors are orthogonal.

The Orthogonal_Mathematic_Vector code listing in § G.1.10 on page 553 contains additional documentation.

12.1.11 Output_Mathematic_Vector Procedure

The Output_Mathematic_Vector procedure writes out a section of a Mathematic Vector to the specified unit.

Calling syntax:

```
call Output (MV, First, Last, Unit, Indent)
```

Input variables:

`MV`	The Mathematic_Vector object to be queried.
`First`	The first location to be output. [Optional]
`Last`	The last location to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]
`Indent`	Number of indentation characters. [Optional]

The Output_Mathematic_Vector code listing in § G.1.11 on page 553 contains additional documentation.

12.1.12 Set_Not_Up_to_Date_Mathematic_Vector Procedure

The Set_Not_Up_to_Date_Mathematic_Vector procedure puts a Mathematic Vector into a "not up-to-date" state. That is, it unsets the values for all of the updated? variables in the Mathematic Vector. It is mainly used internally to the Mathematic Vector class, but could also be used externally to force recalculation of norms, extrema, etc.

Calling syntax:

```
call Set_Not_Up_to_Date (MV)
```

Input/Output variable:

`MV`	The Mathematic_Vector object to be set.

The Set_Not_Up_to_Date_Mathematic_Vector code listing in § G.1.12 on page 557 contains additional documentation.

12.1.13 Set_Values_Mathematic_Vector Procedure

The Set_Values_Mathematic_Vector procedure sets the values for the Mathematic Vector. Note that if the single value form of this procedure is used, then the same number of procedure calls must take place on every processor – this is required by internal verification calls that use global communication. If a different number of values are to be set on each processor, then the vector form should be used.

Calling syntax:

```
MV = Value                                   ,
MV = Values                                  ,
call Set_Value (MV, Value)                   ,
call Set_Values (MV, Value, Row, Global)     ,
call Set_Values (MV, Values, Rows, Global)   or
call Set_Values (MV, Values)
```

Input variable:

`Values`	A vector of values for the Mathematic_Vector object. If the Rows variable is not present, Values must be the same size as the Mathematic_Vector. Otherwise, Values must have a size smaller than or equal to the size of the Mathematic_Vector.
`Rows`	An integer vector of rows for the Values vector. [optional]
`Global`	A toggle between using global or local (on this PE) indices for the Rows variable. Default is true (use global). [optional]
`Value`	A single value for the Mathematic_Vector object. The Row variable must be present for the Set_Values form of the procedure. For the Set_Value or assignment form, all vector values are set to the same input value (often zero).
`Row`	The row index for the Value variable.

Input/Output variable:

`MV`	The Mathematic_Vector object to be set.

The Set_Values_Mathematic_Vector code listing in § G.1.13 on page 557 contains additional documentation.

12.1.14 Update_DV_Mathematic_Vector Procedure

The Update_DV_Mathematic_Vector procedure updates the Distributed Vector inside of a Mathematic Vector from the Values objects. If the DV is already updated, no work is done.

Calling syntax:

```
call Update_DV (MV)
```

Input/Output variable:

`MV`	The Mathematic_Vector object to be set.

The Update_DV_Mathematic_Vector code listing in § G.1.14 on page 561 contains additional documentation.

12.2 ELL_Matrix Class

The ELL_Matrix Class is used to describe a matrix in the CÆSAR Code Package. The storage format is ELL, a common storage format originally used by the ELLPACK package[1]. In this storage format, two rectangular arrays are used to store the matrix. The arrays have the same number of rows as the original matrix, but only have as many columns as the maximum number of nonzeros on a row of the original matrix. One of the arrays holds the matrix entries, and the other array holds the column numbers from the original matrix.

As an example, this original matrix:

$$\text{Full Matrix} = \begin{bmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 0 & 4 & 5 & 0 & 6 & 0 \\ 7 & 0 & 8 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 7 & 6 \\ 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 3 & 0 \end{bmatrix} \tag{12.1}$$

which has a maximum number of nonzeros per row of 3, would be stored as

$$\text{Values} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 8 & 7 & 6 \\ 5 & 0 & 0 \\ 4 & 3 & 0 \end{bmatrix} , \quad \text{Columns} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 5 \\ 1 & 3 & 5 \\ 2 & 5 & 6 \\ 3 & 0 & 0 \\ 3 & 5 & 0 \end{bmatrix} \tag{12.2}$$

This storage format is most efficient when every row in the original matrix has the same number of nonzeros. It is somewhat easier to work with than the compressed sparse row (CSR) format, but it can use more memory if the matrix has an uneven number of nonzeros per row.

In addition to this, the ELL_Matrix Class is a parallel data structure. The row dimension in the `Values` and `Columns` arrays is distributed across the processors.

The ELL_Matrix methods section in § 16.2 on page 190 describes the methods used in the ELL_Matrix Class.

ELL_Matrix public procedures:

Fundamental procedures	
`Initialize`	Initializes an ELL_Matrix object.
`Finalize`	Finalizes an ELL_Matrix object.
`Valid_State`	Returns false iff an ELL_Matrix object is in an invalid state.
`Initialized`	Returns true iff an ELL_Matrix object has been initialized.
Operations	
`Add_Values`	Increments the values of an ELL_Matrix object.
`Average`	Returns the arithmetic mean of the ELL_Matrix object.
`First_Row_PE`	Returns the global number of the first row on this PE of the ELL_Matrix object.
`Frobenius_Norm`	Returns the Frobenius norm of the ELL_Matrix object.
`Get_Columns`	Returns the column locations for an ELL_Matrix object (also has an assignment interface).
`Get_Values`	Returns the values for an ELL_Matrix object (also has an assignment interface).
`Infinity_Norm`	Returns the infinity norm of the ELL_Matrix object.
`Last_Row_PE`	Returns the global number of the last row on this PE of the ELL_Matrix object.
`MatVec`	Returns the global matrix-vector product of an ELL_Matrix object and a Mathematic_Vector object.

[1] `\http://www.cs.purdue.edu/ellpack/`

`Max_Nonzeros`	Returns the maximum number of nonzeros of the ELL_Matrix object (the array storage dimension for the columns).
`Maximum`	Returns the maximum of the ELL_Matrix object.
`Mean`	Returns the mean value of the ELL_Matrix object.
`Minimum`	Returns the minimum of the ELL_Matrix object.
`Name`	Returns the name of the ELL_Matrix object.
`Norm`	Returns the Frobenius norm of the ELL_Matrix object.
`NColumns`	Returns the number of columns of the ELL_Matrix object.
`NRows_PE`	Returns the number of rows on this PE of the ELL_Matrix object.
`NRows_Total`	Returns the total number of rows of the ELL_Matrix object.
`One_Norm`	Returns the one norm of the ELL_Matrix object.
`Output`	Writes out the ELL_Matrix object.
`Read`	Reads in an ELL_Matrix object from a Harwell-Boeing formatted file. Also reads in Mathematic_Vector objects from the file.
`Read_Harwell_Boeing`	Reads in an ELL_Matrix object from a Harwell-Boeing formatted file. Also reads in Mathematic_Vector objects from the file.
`Residual`	Calculates the residual vector ($r = Ax - b$) for a linear system.
`Set_0_Diagonal_to_1`	Sets the diagonal of the matrix to unity if all entries on a row are equal to zero.
`Set_Not_Up_to_Date`	Puts an ELL Matrix into a "not up-to-date" state. Can be used externally to force recalculation of norms, extrema, etc.
`Set_Value`	Sets all values to a specified scalar (often zero) for an ELL_Matrix object. Does not modify sparsity pattern or any other internal structure.
`Set_Values`	Sets the values for an ELL_Matrix object.
`Sum`	Returns the sum of the ELL_Matrix object.
`Total`	Returns the sum of the ELL_Matrix object.
`Two_Norm_Estimate`	Returns an estimate of the two norm of the ELL_Matrix object.
`Two_Norm_Range`	Returns the range of possible values for the two norm of the ELL_Matrix object.

ELL_Matrix public defined types:

ELL_Matrix type	
`Average`	Global average of the ELL Matrix.
`Average_is_Updated`	Updated? toggle for Average.
`Column_Structure`	Column base structure for the ELL Matrix.
`Columns`	Array of column numbers for the ELL Matrix.
`Dimensionality`	Number of dimensions for the ELL Matrix (always unity).
`Index`	Data Index that is used for matvecs.
`Index_is_Updated`	Updated? toggle for Index.
`Index_Match_Number`	Index Match Number that is used for matvecs.
`Frobenius_Norm`	Frobenius norm of the ELL Matrix.
`Frobenius_Norm_is_Updated`	Updated? toggle for Frobenius_Norm.
`Infinity_Norm`	Infinity norm of the ELL Matrix.
`Infinity_Norm_is_Updated`	Updated? toggle for Infinity_Norm.
`Initialized`	Initialization status.
`Max_Nonzeros`	Maximum number of nonzero columns on a row in the matrix.
`Maximum`	Global maximum of the ELL Matrix.
`Maximum_is_Updated`	Updated? toggle for Maximum.
`Minimum`	Global minimum of the ELL Matrix.
`Minimum_is_Updated`	Updated? toggle for Minimum.
`Name`	The name for this variable (especially useful in a vector of ELL Matrices).
`One_Norm`	One norm of the ELL Matrix.
`One_Norm_is_Updated`	Updated? toggle for One_Norm.
`OV`	A vector of Overlapped Vectors that is used for matvecs.
`Row_Structure`	Row base structure for the ELL Matrix.

`Sum`	Global sum of the ELL Matrix.
`Sum_is_Updated`	Updated? toggle for Sum.
`Two_Norm_Estimate`	An estimate of the two norm of the ELL Matrix, taken to be the midpoint of the range.
`Two_Norm_is_Updated`	Updated? toggle for Two_Norm.
`Two_Norm_Range`	The possible range of the two norm of the ELL Matrix.
`Values`	Values for the ELL Matrix.

The ELL_Matrix Class code listing in § G.2 on page 565 contains additional documentation. The ELL_Matrix Class also contains a Unit Test Program which is listed in § G.2.16 on page 610.

12.2.1 Initialize_ELL_Matrix Procedure

The Initialize_ELL_Matrix procedure allocates and initializes an ELL_Matrix object.

Calling syntax:

```
call Initialize (ELLM, Max_Nonzeros, Row_Structure, Column_Structure, Name, status)
```

Input variables:

`ELLM`	The ELL_Matrix object to be initialized.
`Max_Nonzeros`	The maximum number of nonzero elements per row.
`Row_Structure`	The Base_Structure giving the row distribution for the ELL_Matrix.
`Column_Structure`	The Base_Structure giving the column distribution for the ELL_Matrix.
`Name`	The name for this variable. [Optional]

Output variables:

`ELLM`	The ELL_Matrix object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_ELL_Matrix code listing in § G.2.1 on page 569 contains additional documentation.

12.2.2 Finalize_ELL_Matrix Procedure

The Finalize_ELL_Matrix procedure deallocates and finalizes an ELL_Matrix object.

Calling syntax:

```
call Finalize (ELLM, status)
```

Input variables:

`ELLM`	The ELL_Matrix object to be finalized.

Output variables:

`ELLM`	The ELL_Matrix object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status ell_matrix.
`i`	Loop variable.

The Finalize_ELL_Matrix code listing in § G.2.2 on page 571 contains additional documentation.

12.2.3 Valid_State_ELL_Matrix Procedure

The Valid_State_ELL_Matrix procedure returns true iff the ELL_Matrix is in a valid state – that is, iff the ELL_Matrix passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(ELLM)
```

Input variables:

`ELLM`	The ELL_Matrix to be checked.

Output variable:

`Valid_State`	True iff the ELL_Matrix is in a valid state.

The Valid_State_ELL_Matrix code listing in § G.2.3 on page 573 contains additional documentation.

12.2.4 Initialized_ELL_Matrix Procedure

The Initialized_ELL_Matrix procedure returns true iff the ELL_Matrix object has been initialized.

Calling syntax:

```
Logical = Initialized(ELLM)
```

Input variable:

`ELLM`	The ELL_Matrix object to be examined.

Output variable:

`Initialized`	True iff the ELL_Matrix object has been initialized.

The Initialized_ELL_Matrix code listing in § G.2.4 on page 575 contains additional documentation.

12.2.5 Add_Values_ELL_Matrix Procedure

The Add_Values_ELL_Matrix procedure increments the values for an ELL Matrix by the specified array of values. It may be called three different ways.

The single value form adds to a single matrix entry. If this form is used, then the same number of procedure calls must take place on every processor – this is required by internal verification calls that use global communication. If a different number of values are to be add on each processor, then one of the other calling forms should be used.

The vector form, with the plural nomenclature, can be used to add to selected values, one per row, in the ELL Matrix. Only the rows listed in the Rows vector are modified.

The array form, with the plural nomenclature, can be used to add to selected values in the ELL Matrix. Only the rows and columns listed in the Rows and Columns vectors are modified.

Calling syntax:

```
call Add_Values (ELLM, Value, Row, Column, Global)       or
call Add_Values (ELLM, Values, Rows, Columns, Global)
```

Input variable:

`Values`	A vector or an array of values to be added to the ELL_Matrix object. Values must have a size smaller than or equal to the size of the ELL_Matrix.
`Rows`	An integer vector of rows to be incremented. This vector may be sized smaller than the row size of the ELL_Matrix object.
`Columns`	A vector or an array of columns for the ELL_Matrix object. Columns must have a size smaller than or equal to the size of the ELL_Matrix.
`Global`	A toggle between using global or local (on this PE) indices for the Rows variables. Default is true (use global). Note that the Columns variables are always global. [optional]
`Value`	A single value to be added to the ELL_Matrix object. The Row and Column variables must be present for this form of the procedure.
`Row`	The row index for the Value variable.
`Column`	The column index for the Value variable.

Input/Output variable:

`ELLM`	The ELL_Matrix object to be incremented.

The Add_Values_ELL_Matrix code listing in § G.2.5 on page 576 contains additional documentation.

12.2.6 Get Value ELL_Matrix Functions

The Get_Value_ELLM functions return values from an ELL_Matrix object. These operations require global communication (except Get_First_Row_PE, Get Last_Row_PE, Get_Max_Nonzeros, Get_Name_ELLM, Get_-NColumns, Get_NRows_PE and Get_NRows_Total), but if called more than once without modifying the object, no global communication is done for the second call.

Calling syntax:

```
Output = Average (ELLM)              ,
Output = First_Row_PE (ELLM)         ,
Output = Frobenius_Norm (ELLM)       ,
Output = Infinity_Norm (ELLM)        ,
Output = Last_Row_PE (ELLM)          ,
Output = Max_Nonzeros (ELLM)         ,
Output = Maximum (ELLM)              ,
Output = Mean (ELLM)                 ,
Output = Minimum (ELLM)              ,
Output = Name (ELLM)                 ,
Output = Norm (ELLM)                 ,
Output = NColumns (ELLM)             ,
Output = NRows_PE (ELLM)             ,
Output = NRows_Total (ELLM)          ,
Output = One_Norm (ELLM)             ,
Output = Sum (ELLM)                  ,
Output = Total (ELLM)                ,
Output = Two_Norm_Estimate (ELLM)    or
Output = Two_Norm_Range (ELLM)
```

Mean is an alternate interface name for the Average Procedure, Norm is an alternate interface name for the Frobenius_Norm Procedure, and Total is an alternate interface name for the Sum Procedure.

The Two Norm is problematic and is not calculated exactly by CÆSAR currently. However, the Two_Norm_-Range function returns limits on the possible values of the Two Norm, and the Two_Norm_Estimate function returns the midpoint of this range.

The First_Row_PE and Last_Row_PE functions return the global numbers for the first and last rows on this PE.

Input variables:

`ELLM` The ELL_Matrix object to be examined.

Output variable:

`Output` For Name, returns a character variable containing the name assigned to the object upon initialization. For First_Row_PE, Last_Row_PE, Max_Nonzeros, NColumns, NRows_PE and NRows_Total, returns an integer variable with the requested number. For all other functions, returns a real variable with the named value for the ELL_Matrix object.

The Get Value ELL_Matrix code listing in § G.2.6 on page 582 contains additional documentation.

12.2.7 Get_Columns_ELL_Matrix Procedure

The Get_Columns_ELL_Matrix procedure accesses the column locations from an ELL Matrix.

Calling syntax:

```
Columns = ELLM                      or
call Get_Columns (Columns, ELLM)
```

Input variable:

`Columns`	The bare naked array of columns from the ELL_Matrix object. Columns must be the same size as the ELL_Matrix.

Input/Output variable:

`ELLM`	The ELL_Matrix object to be accessed.

The Get_Columns_ELL_Matrix code listing in § G.2.7 on page 585 contains additional documentation.

12.2.8 Get_Values_ELL_Matrix Procedure

The Get_Values_ELL_Matrix procedure accesses the values from an ELL Matrix.

Calling syntax:

```
Values = ELLM                        or
call Get_Values (Values, ELLM)
```

Input variable:

`Values`	The bare naked array of values from the ELL_Matrix object. Values must be the same size as the ELL_Matrix.

Input/Output variable:

`ELLM`	The ELL_Matrix object to be accessed.

The Get_Values_ELL_Matrix code listing in § G.2.8 on page 586 contains additional documentation.

12.2.9 MatVec_ELL_Matrix Procedure

The MatVec procedure calculates the global matrix-vector product of an ELL Matrix and a Mathematic Vector. It has some "smart" features:

- Overlapped_Vectors are used to store off-processor elements between MatVecs.
- MatVecs between a matrix (that may have changed) and a vector that hasn't changed do not do any global communication.
- The above is true for up to four (current number, could be expanded) different matrices, with different sparsity patterns.

Calling syntax:

```
call MatVec (ELLM, MV_in, MV_out)
```

Input variables:

`ELLM`	An ELL_Matrix object to be multiplied.
`MV_in`	A Mathematic_Vector object to be multiplied. The Structure of the Mathematic_-Vector must be the same as the Column_Structure of the ELL_Matrix.
`MV_out`	Mathematic_Vector object for output, but must have a Valid_State and the same Structure as the Row_Structure of ELLM on input.

Output variable:

`MV_out`	Mathematic_Vector object result of multiplying the ELL_Matrix object by the Mathematic_Vector object. The resultant Mathematic_Vector will have the same Structure as the Row_Structure of the ELL_Matrix.

The MatVec_ELL_Matrix code listing in § G.2.9 on page 587 contains additional documentation.

12.2.10 Output_ELL_Matrix Procedure

The Output_ELL_Matrix procedure writes out a section of a ELL Matrix to the specified unit.

Calling syntax:

```
call Output (ELLM, Row_First, Row_Last, Unit, Indent)
```

Input variables:

`ELLM`	The ELL_Matrix object to be queried.
`Row_First`	The first row location to be output. [Optional]
`Row_Last`	The last row location to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]
`Indent`	Number of indentation characters. [Optional]

The Output_ELL_Matrix code listing in § G.2.10 on page 589 contains additional documentation.

12.2.11 Read_Harwell_Boeing_ELL_Matrix Procedure

The Read_Harwell_Boeing_ELL_Matrix procedure reads and initializes an ELL_Matrix object from a Harwell-Boeing formatted file. It will also read in and initialize a Mathematic_Vector object for a right-hand side vector, an initial guess vector and/or an exact solution vector.

Calling syntax:

```
call Read_Harwell_Boeing (ELLM, RHS_MV, Solution_MV, Guess_MV, Row_Structure, Column_Structure, Unit, status)
```

Input variables:

`Unit`	The logical unit for input, which defaults to 5. [Optional]

Output variables:

`ELLM`	The ELL_Matrix object to be read in and initialized.
`RHS_MV`	The right-hand side Mathematic_Vector object to be read in. [Optional]
`Solution_MV`	The exact solution Mathematic_Vector object to be read in. [Optional]
`Guess_MV`	The initial guess Mathematic_Vector object to be read in. [Optional]
`Row_Structure`	The Base_Structure giving the row distribution for the ELL_Matrix.
`Column_Structure`	The Base_Structure giving the column distribution for the ELL_Matrix.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Read_Harwell_Boeing_ELL_Matrix code listing in § G.2.11 on page 593 contains additional documentation.

12.2.12 Residual_ELL_Matrix Procedure

The Residual procedure calculates the global residual vector of a linear system represented by an ELL Matrix and a two Mathematic Vectors.

Calling syntax:

```
call Residual (Residual_MV, A_ELLM, X_MV, B_MV)
```

Input variables:

`A_ELLM`	An ELL_Matrix object to be multiplied.
`X_MV`	A Mathematic_Vector object to be multiplied. The Structure of the Mathematic_Vector must be the same as the Column_Structure of the ELL_Matrix.
`B_MV`	Mathematic_Vector object to be subtracted from the MatVec. The Structure of the Mathematic_Vector must be the same as the Row_Structure of A_ELLM.

Output variable:

`Residual_MV`	Mathematic_Vector object result of multiplying A_ELLM by X_MV and subtracting B_MV. The resultant Mathematic_Vector will have the same Structure as the Row_Structure of the ELL_Matrix.

The Residual_ELL_Matrix code listing in § G.2.12 on page 600 contains additional documentation.

12.2.13 Set_0_Diagonal_to_1_ELL_Matrix Procedure

The Set_0_Diagonal_to_1_ELL_Matrix procedure sets the diagonal of a matrix to unity if the equation for that diagonal is missing. That is, if all entries on a row of a matrix are zero (both the values and the column numbers) then the diagonal is set to one.

Calling syntax:

```
call Set_0_Diagonal_to_1 (ELLM)
```

Input/Output variable:

`ELLM`	The ELL_Matrix object to be set.

The Set_0_Diagonal_to_1_ELL_Matrix code listing in § G.2.13 on page 601 contains additional documentation.

12.2.14 Set_Not_Up_to_Date_ELL_Matrix Procedure

The Set_Not_Up_to_Date_ELL_Matrix procedure puts an ELL Matrix into a "not up-to-date" state. That is, it unsets the values for all of the updated? variables in the ELL Matrix. It is mainly used internally to the ELL Matrix class, but could also be used externally to force recalculation of norms, extrema, etc.

Calling syntax:

```
call Set_Not_Up_to_Date (ELLM)
```

Input/Output variable:

`ELLM`	The ELL_Matrix object to be set.

The Set_Not_Up_to_Date_ELL_Matrix code listing in § G.2.14 on page 602 contains additional documentation.

12.2.15 Set_Values_ELL_Matrix Procedure

The Set_Values_ELL_Matrix procedure sets the values for the ELL Matrix. It may be called five different ways.

The single value form for Set_Value replaces a single matrix entry. If this form is used, then the same number of procedure calls must take place on every processor – this is required by internal verification calls that use global communication. If a different number of values are to be set on each processor, then one of the other calling forms should be used.

The single value form for Set_Value (also with an assignment interface) sets all the values of a matrix to a single scalar value (often zero). It only sets values which have nonzero column indices, and it does not alter the sparsity pattern or any of the other matrix internals.

The vector form, with the Rows variable, can be used to replace selected values, one per row, in the ELL Matrix. Only the rows listed in the Rows vector are modified.

The array form, with the Rows variable, can be used to replace selected values in the ELL Matrix. Only the rows listed in the Rows vector are modified.

The array form, without the Rows variable, can be used to replace the entire contents of the ELL Matrix. The Values and Columns variables must be exactly the same size as the ELL Matrix variables with the same name.

Calling syntax:

```
ELLM = Value                                              ,
call Set_Value (ELLM, Value)                              ,
call Set_Values (ELLM, Value, Row, Column, Global)        ,
call Set_Values (ELLM, Values, Rows, Columns, Global)     or
call Set_Values (ELLM, Values, Columns)
```

Input variable:

`Values`	A vector or an array of values for the ELL_Matrix object. If the Rows variable is not present, Values must be the same size as the ELL_Matrix. Otherwise, Values must have a size smaller than or equal to the size of the ELL_Matrix.
`Rows`	An integer vector of rows to be replaced. This vector may be sized smaller than the row size of the ELL_Matrix object. [optional]
`Columns`	A vector or an array of columns for the ELL_Matrix object. If the Rows variable is not present, Columns must be the same size as the ELL_Matrix. Otherwise, Columns must have a size smaller than or equal to the size of the ELL_Matrix.
`Global`	A toggle between using global or local (on this PE) indices for the Rows variables. Default is true (use global). Note that the Columns variables are always global. [optional]
`Value`	A single value for the ELL_Matrix object. The Row and Column variables must be present for the Set_Values form of the procedure. For the Set_Value or assignment form, all matrix values are set to the same input value (often zero).

`Row`	The row index for the Value variable.
`Column`	The column index for the Value variable.

Input/Output variable:

`ELLM`	The ELL_Matrix object to be set.

The Set_Values_ELL_Matrix code listing in § G.2.15 on page 603 contains additional documentation.

12.3 Solver Class

The Solver Class is used to describe a Solver in the CÆSAR Code Package. A Solver interacts with the other Solver class by blah blah blah.

The Solver methods section in § 16.3 on page 191 describes the methods used in the Solver Class.

Solver public procedures:

Fundamental procedures	
`Initialize`	Initializes a Solver object.
`Finalize`	Finalizes a Solver object.
`Valid_State`	Returns false iff a Solver object is in an invalid state.
`Initialized`	Returns true iff a Solver object has been initialized.
Operations	
`Procedure`	Description.
`Name`	Returns the name of the Solver object.
`Output`	Writes out the Solver object.

Solver public defined types:

solver type

`Variable`	Description. [Units]

solver type

`Variable`	Description. [Units]
`Initialized`	Initialization status.

Solver public variables:

variable type

`Variable`	Description. [Units]

variable type

`Variable`	Description. [Units]

The Solver Class code listing in § G.3 on page 616 contains additional documentation. The Solver Class also contains a Unit Test Program which is listed in § G.3.8 on page 629.

12.3.1 Initialize_Solver Procedure

The Initialize_Solver procedure allocates and initializes a Solver object.

Calling syntax:

```
call Initialize (Solver, Package, status)
```

Input variables:

`Solver`	The Solver object to be initialized.
`Package`	The linear algebra package to call.

Output variables:

`Solver`	The Solver object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Solver code listing in § G.3.1 on page 618 contains additional documentation.

12.3.2 Finalize_Solver Procedure

The Finalize_Solver procedure deallocates and finalizes a Solver object.

Calling syntax:

```
call Finalize (Solver, status)
```

Input variables:

`Solver`	The Solver object to be finalized.

Output variables:

`Solver`	The Solver object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Solver code listing in § G.3.2 on page 619 contains additional documentation.

12.3.3 Valid_State_Solver Procedure

The Valid_State_Solver procedure returns true iff the Solver is in a valid state – that is, iff the Solver passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Solver)
```

Input variables:

`Solver`	The Solver to be checked.

Output variable:

`Valid_State`	True iff the Solver is in a valid state.

The Valid_State_Solver code listing in § G.3.3 on page 621 contains additional documentation.

12.3.4 Initialized_Solver Procedure

The Initialized_Solver procedure returns true iff the Solver object has been initialized.

Calling syntax:

```
Logical = Initialized(Solver)
```

Input variable:

`Solver`	The Solver object to be examined.

Output variable:

`Initialized`	True iff the Solver object has been initialized.

The Initialized_Solver code listing in § G.3.4 on page 622 contains additional documentation.

12.3.5 Set_Solver_Variable Procedure

The Set_Solver_Variable procedure sets a Solver internal variable.

Calling syntax:

```
call Set (Solver, Variable, Value)
```

Input variables:

`Solver`	The Solver object to be set.

`Variable` The variable within the Solver object to be set. Valid variables, their types and descriptions are:

Epsilon	real	Solver solution tolerance.
Maximum_Iterations	integer	For iterative solvers, the maximum number of iterations allowed.
Stopping_Test	character	The stopping test. Currently the only stopping test available is '——r——/——b—— ¡ Eps'.
LAMG_levout	integer	The output level variable for the LAMG package.

`Value` The value to set the variable to, which is required to be the proper type (real, integer, character, logical) for a given variable.

Output variables:

`Solver` Modified Solver object.

The Set_Solver_Variable code listing in § G.3.5 on page 622 contains additional documentation.

12.3.6 Convert_ELL_to_LAMG Procedure

The Convert_ELL_to_LAMG procedure converts an ELL Matrix object to the LAMG matrix storage format. While converting to the LAMG CSR (compressed storage row) format, the zero entries in the ELL Matrix are removed, which will result in faster operation. This function is only available if CÆSAR has been compiled with LAMG.

Calling syntax:

```
call Convert (LAMG_Matrix, ELLM, LAMG_Communicator, LAMG_Options, status)
```

Input variables:

`ELLM`	The ELL_Matrix object to be converted.
`LAMG_Communicator`	The Communicator object for the LAMG package.
`LAMG_Options`	The Options object for the LAMG package.

Output variables:

`LAMG_Matrix`	The matrix in LAMG format.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

Status variables	
`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.
`LAMG_Status`	LAMG status.
Matrix variables	
`NNonzeros_PE`	Number of nonzeros on this PE.
`Columns_BNA`	Matrix columns bare naked array.
`Values_BNA`	Matrix values bare naked array.
`LAMG_BR_Matrix`	LAMG block-row format matrix.

Counter variables

`BR_Location`	Location in the block-row matrix.
`ELL_Location`	Location in the ELL matrix.
`row`	Row loop counter.

The Convert_ELL_to_LAMG code listing in § G.3.6 on page 623 contains additional documentation.

12.3.7 Solve Procedure

The Solve procedure sets a Solver internal variable.

Calling syntax:

```
call Set (Solver, Variable, Value)
```

Input variables:

Subtype variables

`Variable` Description. [Units]

Subtype variables

`Variable` Description. [Units]

Output variables:

`Variable` Description. [Units]

Internal variables:

`Variable` Description. [Units]

The Solve code listing in § G.3.7 on page 626 contains additional documentation.

Chapter 13

Equation Module

13.1 Monomial Class

The Monomial Class is used to describe a Monomial in the CÆSAR Code Package. A Monomial interacts with the other Monomial class by blah blah blah.

The Monomial methods section in § ?? on page ?? describes the methods used in the Monomial Class.

Monomial public procedures:

Fundamental procedures	
`Initialize`	Initializes a Monomial object.
`Finalize`	Finalizes a Monomial object.
`Valid_State`	Returns false iff a Monomial object is in an invalid state.
`Initialized`	Returns true iff a Monomial object has been initialized.
Operations	
`Add_to_Matrix_Equation_Monomial`	Description.
`Name`	Returns the name of the Monomial object.
`Output`	Writes out the Monomial object.

Monomial public defined types:

Monomial type

`Variable`	Description. [Units]

monomial type

`Variable`	Description. [Units]
`Initialized`	Initialization status.

Monomial public variables:

variable type

`Variable`	Description. [Units]

variable type

`Variable`	Description. [Units]

The Monomial Class code listing in § H.1 on page 631 contains additional documentation. The Monomial Class also contains a Unit Test Program which is listed in § H.1.8 on page 645.

13.1.1 Initialize_Monomial Procedure

The Initialize_Monomial procedure allocates and initializes a Monomial object.

Calling syntax:

```
call Initialize (Monomial, Coefficient, Exponent, Phi_k_MV, Locus, Mesh, Name, Derivative, Delta_t, Phi_n_MV, Variable, Equation, NEquations, status)
```

Input variables:

Subtype variables

`Monomial`	The Monomial object to be initialized.
`Coefficient`	A vector of coefficents for the monomial.
`Exponent`	The degree (exponent) of the monomial.
`Phi_k_MV`	A Mathematic Vector of the independent variable that this monomial is based on, the previous iterate value.
`Locus`	The location for the monomial (e.g. Cells, Nodes or Faces)
`Mesh`	The Mesh object that this monomial is based on.
`Name`	The name for this variable. [Optional]
`Derivative`	Logical toggle for whether to treat this term as a time derivative. [Optional, default is false]
`Delta_t`	The time step size. [Optional, default is 1]
`Phi_n_MV`	A Mathematic Vector of the independent variable that this monomial is based on, the past time step value. This is used for time derivative monomials only, and if left out then the last iterate value, Phi_k_MV, is used instead. [Optional]
`Variable`	The variable number for this term. [Optional]
`Equation`	The equation number for this term. [Optional]
`NEquations`	The total number of equations in this system. [Optional]

Output variables:

`Monomial`	The Monomial object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Monomial code listing in § H.1.1 on page 634 contains additional documentation.

13.1.2 Finalize_Monomial Procedure

The Finalize_Monomial procedure deallocates and finalizes a Monomial object.

Calling syntax:

```
call Finalize (Monomial, status)
```

Input variables:

`Monomial`	The Monomial object to be finalized.

Output variables:

`Monomial`	The Monomial object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`Variable`	Description. [Units]
`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Monomial code listing in § H.1.2 on page 636 contains additional documentation.

13.1.3 Valid_State_Monomial Procedure

The Valid_State_Monomial procedure returns true iff the Monomial is in a valid state – that is, iff the Monomial passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Monomial)
```

Input variables:

`Monomial`	The Monomial to be checked.

Output variable:

`Valid_State`	True iff the Monomial is in a valid state.

Internal variables:

`Variable`	Description. [Units]

The Valid_State_Monomial code listing in § H.1.3 on page 638 contains additional documentation.

13.1.4 Initialized_Monomial Procedure

The Initialized_Monomial procedure returns true iff the Monomial object has been initialized.

Calling syntax:

```
Logical = Initialized(Monomial)
```

Input variable:

`Monomial`	The Monomial object to be examined.

Output variable:

`Initialized`	True iff the Monomial object has been initialized.

The Initialized_Monomial code listing in § H.1.4 on page 639 contains additional documentation.

13.1.5 Add_to_Matrix_Equation_Monomial Procedure

The Add_to_Matrix_Equation_Monomial procedure adds the linearized version of the a monomial or the time derivative of a monomial to the specified matrix equation. It currently assumes that there is a one-to-one correspondence between the Monomial locus (e.g. Cells) and the equation and variable numbering, and furthermore assumes that the locus is Cells.

The linearization for a non-time-derivative monomial is as follows:

$$\int_V C\Phi^E \, dV \approx CV\left[\Phi_k^E + E\Phi_k^{E-1}\left(\Phi_{k+1} - \Phi_k\right)\right] \tag{13.1}$$

$$= CVE\Phi_k^{E-1}\Phi_{k+1} + CV\left(1 - E\right)\Phi_k^E \tag{13.2}$$

The linearization for a time-derivative monomial is as follows:

$$\int_V C\frac{\partial}{\partial t}\left(\Phi^E\right) \, dV \approx \frac{CV}{\Delta t}\left[\Phi_k^E + E\Phi_k^{E-1}\left(\Phi_{k+1} - \Phi_k\right) - \Phi_n^E\right] \tag{13.3}$$

$$= \frac{CVE}{\Delta t}\Phi_k^{E-1}\Phi_{k+1} + \frac{CV}{\Delta t}\left[\left(1 - E\right)\Phi_k^E - \Phi_k^n\right] \tag{13.4}$$

where Φ_k is the kth iterate, Φ_n is the past time step value, and Φ_{k+1} approaches Φ_{n+1} with successive iterations.

Calling syntax:

```
call Add_to_Matrix_Equation (Monomial, ELLM, RHS_MV)
```

Input variables:

`Monomial`	The Monomial Term to be added.
`ELLM`	The ELL Matrix to be incremented.
`RHS_MV`	The right-hand side Mathematic Vector to be incremented.

Output variables:

`ELLM`	The ELL Matrix that has been incremented.
`RHS_MV`	The right-hand side Mathematic Vector that has been incremented.

Internal variables:

`Matrix_Columns`	Columns for the added matrix values.
`Matrix_Rows`	Rows for the added matrix values.
`Matrix_Values`	Calculated values to be added to the matrix or vector.

The Add_to_Matrix_Equation_Monomial code listing in § H.1.5 on page 640 contains additional documentation.

13.1.6 Get Value Monomial Functions

The Get_Value_Monomial functions return values from a Monomial object.

Calling syntax:

```
Output = Locus (Monomial)    or
Output = Name (Monomial)
```

Input variables:

`Monomial`	The Monomial object to be examined.

Output variable:

`Output`	Returns a character variable containing the name or locus assigned to the object upon initialization.

The Get Value Monomial code listing in § H.1.6 on page 642 contains additional documentation.

13.1.7 Output_Monomial Procedure

The Output_Monomial procedure writes out a section of a Mathematic Vector to the specified unit.

Calling syntax:

```
call Output (Monomial, First, Last, Unit, Indent, status)
```

Input variables:

`Monomial`	The Monomial object to be queried.
`First`	The first location to be output. [Optional]
`Last`	The last location to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]
`Indent`	Number of indentation characters. [Optional]

Output variables:

`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`Coefficient_MV`	Temporary Mathematic Vector for the Coefficient.
`Phi_k_MV`	Temporary Mathematic Vector for Phi_k.
`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Output_Monomial code listing in § H.1.7 on page 643 contains additional documentation.

13.2 Ortho_Diffusion Class

The Ortho_Diffusion Class is used to describe an Orthogonal Diffusion equation term in the CÆSAR Code Package.

The Ortho_Diffusion methods section in § ?? on page ?? describes the methods used in the Ortho_Diffusion Class.

Ortho_Diffusion public procedures:

Fundamental procedures	
`Initialize`	Initializes an Ortho_Diffusion object.
`Finalize`	Finalizes an Ortho_Diffusion object.
`Valid_State`	Returns false iff an Ortho_Diffusion object is in an invalid state.
`Initialized`	Returns true iff an Ortho_Diffusion object has been initialized.
Operations	
`Add_to_Matrix_Equation`	Adds the linearized version of the Ortho_Diffusion Term to the specified matrix equation.
`Evaluate`	Evaluates the Ortho_Diffusion Term ($\int_V \overrightarrow{\nabla} \cdot D \overrightarrow{\nabla} \Phi \, dV$) at the cell centers.
`Evaluate_Gradient_Cells`	Evaluates the gradient of Phi at the cell centers using the linearization of the Ortho_Diffusion Term.
`Locus`	Returns the locus of the Ortho_Diffusion object.
`Name`	Returns the name of the Ortho_Diffusion object.
`Output`	Writes out the Ortho_Diffusion object.

Ortho_Diffusion public defined types:

Ortho_Diffusion type	
`Boundary_Condition`	The boundary condition flag for each face of each cell.
`Coefficient`	The diffusion coefficient, defined on the cells for now.
`Equation`	The equation to which this term will be added.
`Initialized`	Initialization status.
`Locus`	Evaluation locus.
`Mesh`	Mesh that this Ortho_Diffusion object is defined on.
`Name`	The name for this variable.
`NEquations`	The number of equations in the entire system.
`Phi`	The independent variable at the linearization value (past time step or iterate).
`Phi_BC`	The boundary condition constant for each face of each cell.
`Structure`	Locus Base_Structure.

Ortho_Diffusion private procedures:

`Get_Harmonic_Diffusion_Coef`	Returns the harmonic diffusion coefficients for each face of each cell.

The Ortho_Diffusion Class code listing in § H.2 on page 648 contains additional documentation. The Ortho_Diffusion Class also contains a Unit Test Program which is listed in § H.2.11 on page 675.

13.2.1 Initialize_Ortho_Diffusion Procedure

The Initialize_Ortho_Diffusion procedure allocates and initializes an Ortho_Diffusion Term object.

Calling syntax:

```
call Initialize (Diff_Term, Coefficient, BC_Faces_of_Cells, Phi_BC_Faces_of_Cells, Phi_MV, Locus, Mesh, Name, Extrapolation, Equation, NEquations, status)
```

Input variables:

Subtype variables	
`Diff_Term`	The Ortho_Diffusion object to be initialized.
`BC_Faces_of_Cells`	Boundary condition flags for this Ortho_Diffusion object.
`Phi_BC_Faces_of_Cells`	Boundary condition constants for this Ortho_Diffusion object.
`Coefficient`	A vector of diffusion coefficents.
`Phi_MV`	An Mathematic Vector of the independent variable that this Ortho_Diffusion object is based on (past time step or iterate value).
`Locus`	The location for the Ortho_Diffusion Term (e.g. Cells, Nodes or Faces)
`Mesh`	The Mesh object that this Ortho_Diffusion Term is based on.
`Name`	The name for this variable. [Optional]
`Extrapolation`	A factor used in the source and vacuum boundary conditions, which effectively sets the extrapolation distance to Coefficient/Extrapolation. [Optional, default is 1/2]
`Equation`	The equation number for this variable. [Optional]
`NEquations`	The total number of equations in this system. [Optional]

Output variables:

`Diff_Term`	The Ortho_Diffusion object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Ortho_Diffusion code listing in § H.2.1 on page 651 contains additional documentation.

13.2.2 Finalize_Ortho_Diffusion Procedure

The Finalize_Ortho_Diffusion procedure deallocates and finalizes an Ortho_Diffusion object.

Calling syntax:

```
call Finalize (Diff_Term, status)
```

Input variables:

`Diff_Term`	The Ortho_Diffusion Term object to be finalized.

Output variables:

`Diff_Term`	The Ortho_Diffusion object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

Variable	Description. [Units]
`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Ortho_Diffusion code listing in § H.2.2 on page 654 contains additional documentation.

13.2.3 Valid_State_Ortho_Diffusion Procedure

The Valid_State_Ortho_Diffusion procedure returns true iff the Ortho_Diffusion Term is in a valid state – that is, iff the Ortho_Diffusion Term passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Diff_Term)
```

Input variables:

`Diff_Term`	The Ortho_Diffusion Term to be checked.

Output variable:

`Valid_State`	True iff the Ortho_Diffusion Term is in a valid state.

The Valid_State_Ortho_Diffusion code listing in § H.2.3 on page 655 contains additional documentation.

13.2.4 Initialized_Ortho_Diffusion Procedure

The Initialized_Ortho_Diffusion procedure returns true iff the Ortho_Diffusion Term object has been initialized.

Calling syntax:

```
Logical = Initialized(Diff_Term)
```

Input variable:

`Diff_Term`	The Ortho_Diffusion Term object to be examined.

Output variable:

`Initialized`	True iff the Ortho_Diffusion Term object has been initialized.

The Initialized_Ortho_Diffusion code listing in § H.2.4 on page 656 contains additional documentation.

13.2.5 Add_to_Matrix_Equation_Ortho_Diffusion Procedure

The Add_to_Matrix_Equation_Ortho_Diffusion procedure adds the linearized version of the Ortho_Diffusion Term to the specified matrix equation. It currently assumes that there is a one-to-one correspondence between the Ortho_Diffusion locus (e.g. Cells) and the equation and variable numbering, and furthermore assumes that the locus is Cells.

Calling syntax:

```
call Add_to_Matrix_Equation (Diff_Term, ELLM, RHS_MV)
```

Input variables:

`Diff_Term`	The Ortho_Diffusion Term to be added.
`ELLM`	The ELL Matrix to be incremented.
`RHS_MV`	The right-hand side Mathematic Vector to be incremented.

Output variables:

`ELLM`	The ELL Matrix that has been incremented.
`RHS_MV`	The right-hand side Mathematic Vector that has been incremented.

Internal variables:

`Matrix_Columns`	Columns for the added matrix values.
`Matrix_Rows`	Rows for the added matrix values.
`Matrix_Values`	Calculated values to be added to the matrix or vector.

The Add_to_Matrix_Equation_Ortho_Diffusion code listing in § H.2.5 on page 657 contains additional documentation.

13.2.6 Evaluate_Gradient_Cells_Ortho_Diffusion Procedure

The Evaluate_Gradient_Cells_Ortho_Diffusion procedure evaluates the gradient of Phi at the cell centers using the specified Ortho_Diffusion Term. The gradient is not necessarily constant across a face, since the diffusion coefficient may change, but the flux is. This procedure calculates the constant flux across a face, then divides by the diffusion coefficient of the cell to get the gradient at each face. Then, averaging opposite faces results in a cell-centered gradient.

Calling syntax:

```
call Evaluate_Gradient_Cells (Diff_Term, Phi_MV, Grad_Cells)
```

Input variables:

`Diff_Term`	The Ortho_Diffusion Term to be evaluated.
`Phi_MV`	The Phi Mathematic Vector to use in the evaluation.

Output variables:

`Grad_Cells`	The gradient of Phi vector evaluated at the cell centers.

Internal variable:

`Grad_dot_N_Faces_of_Cells`	The gradient of Phi dotted with the unit normal at each face of each cell.

The Evaluate_Gradient_Cells_Ortho_Diffusion code listing in § H.2.6 on page 661 contains additional documentation.

13.2.7 Evaluate_Ortho_Diffusion Procedure

The Evaluate_Ortho_Diffusion procedure evaluates an Ortho_Diffusion Term, that is, the $-\overrightarrow{\nabla} \cdot D \overrightarrow{\nabla} \Phi$ term, in its discretized form, at the cell centers.

Calling syntax:

```
call Evaluate (Diff_Term, Phi_MV, Div_D_Grad_Phi_Cells)
```

Input variables:

`Diff_Term`	The Ortho_Diffusion Term to be evaluated.
`Phi_MV`	The Phi Mathematic Vector to use in the evaluation.

Output variables:

`Div_D_Grad_Phi_Cells`	The value of minus the divergence of D times the gradient of Phi for each cell.

The Evaluate_Ortho_Diffusion code listing in § H.2.7 on page 665 contains additional documentation.

13.2.8 Get_Harmonic_Diffusion_Coef_Ortho_Diffusion Procedure

The Get_Harmonic_Diffusion_Coef_Ortho_Diffusion procedure calculates the harmonic average of the diffusion coefficient at each face of each cell.

Calling syntax:

```
call Get_Harmonic_Diffusion_Coef ( Harmonic_Diffusion_Coef_F_of_C, Diff_Term)
```

Input variables:

`Diff_Term`	The Ortho_Diffusion Term to be queried.

Output variables:

`Harmonic_Diffusion_Coef_F_of_C`	The harmonic average diffusion coefficient evaluated at each face of each cell on this PE.

The Get_Harmonic_Diffusion_Coef_Ortho_Diffusion code listing in § H.2.8 on page 669 contains additional documentation.

13.2.9 Get Value Ortho_Diffusion Functions

The Get_Value_Ortho_Diffusion functions return values from an Ortho_Diffusion Term object.

Calling syntax:

```
Output = Locus (Diff_Term)    or
Output = Name (Diff_Term)
```

Input variables:

`Diff_Term`	The Ortho_Diffusion Term object to be examined.

Output variable:

`Output`	Returns a character variable containing the name or locus assigned to the object upon initialization.

The Get Value Ortho_Diffusion code listing in § H.2.9 on page 672 contains additional documentation.

13.2.10 Output_Ortho_Diffusion Procedure

The Output_Ortho_Diffusion procedure writes out a section of a Ortho_Diffusion object to the specified unit.

Calling syntax:

```
call Output (Diff_Term, First, Last, Unit, Indent, status)
```

Input variables:

`Diff_Term`	The Ortho_Diffusion object to be queried.
`First`	The first location to be output. [Optional]
`Last`	The last location to be output. [Optional]
`Unit`	The logical unit for output, which defaults to 6. [Optional]
`Indent`	Number of indentation characters. [Optional]

Output variables:

`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`Coefficient_MV`	Temporary Mathematic Vector for the Coefficient.
`Phi_MV`	Temporary Mathematic Vector for Phi.
`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Output_Ortho_Diffusion code listing in § H.2.10 on page 673 contains additional documentation.

Chapter 14

Mesh Module

14.1 Multi_Mesh Class

The Multi_Mesh Class is used to describe a mesh in the CÆSAR Code Package.

The following is incomplete documentation – or rather documentation in the middle of being written.

Dims	Uniformity	Orthogonality	Structure	AMR	Cell-Shape	Geometry
1	Uniform	Orthogonal	Structured	no	Segmented	x-Cartesian
1	Uniform	Orthogonal	Structured	no	Segmented	r-Cylindrical
1	Uniform	Orthogonal	Structured	no	Segmented	r-Spherical
1	Nonuniform	Orthogonal	Structured	no	Segmented	x-Cartesian
1	Nonuniform	Orthogonal	Structured	no	Segmented	r-Cylindrical
1	Nonuniform	Orthogonal	Structured	no	Segmented	r-Spherical
1	Nonuniform	Orthogonal	Unstructured	no	Segmented	x-Cartesian
1	Nonuniform	Orthogonal	Unstructured	no	Segmented	r-Cylindrical
1	Nonuniform	Orthogonal	Unstructured	no	Segmented	r-Spherical
2	Uniform	Orthogonal	Structured	no	Quadrilateral	xy-Cartesian
2	Uniform	Orthogonal	Structured	no	Quadrilateral	rz-Cylindrical
2	Nonuniform	Orthogonal	Structured	no	Quadrilateral	xy-Cartesian
2	Nonuniform	Orthogonal	Structured	no	Quadrilateral	rz-Cylindrical
2	Nonuniform	Nonorthogonal	Structured	no	Quadrilateral	xy-Cartesian
2	Nonuniform	Nonorthogonal	Structured	no	Quadrilateral	rz-Cylindrical
2	Uniform	Orthogonal	Structured	yes	Quadrilateral	xy-Cartesian
2	Uniform	Orthogonal	Structured	yes	Quadrilateral	rz-Cylindrical
2	Nonuniform	Orthogonal	Structured	yes	Quadrilateral	xy-Cartesian
2	Nonuniform	Orthogonal	Structured	yes	Quadrilateral	rz-Cylindrical
2	Nonuniform	Nonorthogonal	Structured	yes	Quadrilateral	xy-Cartesian
2	Nonuniform	Nonorthogonal	Structured	yes	Quadrilateral	rz-Cylindrical
2	Nonuniform	Nonorthogonal	Unstructured	no	Quadrilateral	xy-Cartesian
2	Nonuniform	Nonorthogonal	Unstructured	no	Quadrilateral	rz-Cylindrical
2	Nonuniform	Nonorthogonal	Unstructured	yes	Quadrilateral	xy-Cartesian
2	Nonuniform	Nonorthogonal	Unstructured	yes	Quadrilateral	rz-Cylindrical
2	Nonuniform	Nonorthogonal	Unstructured	no	Triangular	xy-Cartesian
2	Nonuniform	Nonorthogonal	Unstructured	no	Triangular	rz-Cylindrical
2	Nonuniform	Nonorthogonal	Unstructured	no	Polygonal	xy-Cartesian
2	Nonuniform	Nonorthogonal	Unstructured	no	Polygonal	rz-Cylindrical
3	Uniform	Orthogonal	Structured	no	Hexahedral	xyz-Cartesian

```
3   Nonuniform  Orthogonal     Structured    no  Hexahedral    xyz-Cartesian
3   Nonuniform  Nonorthogonal  Structured    no  Hexahedral    xyz-Cartesian
3   Uniform     Orthogonal     Structured    yes Hexahedral    xyz-Cartesian
3   Nonuniform  Orthogonal     Structured    yes Hexahedral    xyz-Cartesian
3   Nonuniform  Nonorthogonal  Structured    yes Hexahedral    xyz-Cartesian
3   Nonuniform  Nonorthogonal  Unstructured  no  Hexahedral    xyz-Cartesian
3   Nonuniform  Nonorthogonal  Unstructured  yes Hexahedral    xyz-Cartesian
3   Nonuniform  Nonorthogonal  Unstructured  no  Tetrahedral   xyz-Cartesian
3   Nonuniform  Nonorthogonal  Unstructured  no  Polyhedral    xyz-Cartesian

In general, treat the mesh as:

*   Nonuniform  Nonorthogonal  Unstructured  *   *             if-check

where *-values are mostly handled inside the parallel data structures.

Implications:

 Uniform meshes must be structured.
 1-D meshes must be orthogonal.
 2-D means "not spherical".
```

Mesh Nomenclature

A mesh consists of a collection of spatial locations or points, called *nodes*, that are connected in various ways to define distinct spatial volumes, called *cells*, and to define surface areas, called *faces*, between the cells and on the boundaries of the mesh.

In a three-dimensional mesh, nodes are zero-dimensional, faces are two-dimensional[1], and cells are three-dimensional.

In a two-dimensional mesh, nodes are zero-dimensional, faces are one-dimensional (with a suppressed second dimension), and cells are two-dimensional (with a suppressed third dimension).

In a one-dimensional mesh, nodes are zero-dimensional, faces are zero-dimensional (with two suppressed dimensions), and cells are one-dimensional (with two additional suppressed dimensions).

The *connectivity* of a mesh refers to the way that nodes are connected to form cells, and the way that cells are connected to each other. The connectivity of a mesh may be specified in several ways, but a common form gives the node numbers surrounding each cell in an array.

The *cell-shape* of a mesh is determined by the maximum allowed number of faces for a cell in the mesh. In some meshes, degenerate cells are allowed which have fewer than the maximum number of faces. For one-dimensional meshes, the only possible cell-shape is *segmented*, which refers to a line segment or bar as the representation of a cell. For two-dimensional meshes, the possible cell-shapes are *triangular*, *quadrilateral*, or *polygonal*. For three-dimensional meshes, the possible cell-shapes are *tetrahedral*, *hexahedral*, or *polyhedral*.

The *geometry* of a mesh refers to both the coordinate system used by the mesh and the common names for the coordinate axes that are not suppressed. With a Cartesian coordinate system, using the common names of x, y, and z for the axes, the possible geometries are x-Cartesian, xy-Cartesian, and xyz-Cartesian. With a cylindrical coordinate system, using the common names of r, θ, and z for the axes, the possible unique geometries are r-Cylindrical, θ-Cylindrical, $r\theta$-Cylindrical, rz-Cylindrical, and $r\theta z$-Cylindrical. With a spherical coordinate system, using the common names of r, θ (azimuth, range of 0-2π), and ϕ (zenith, range of 0-π) for the axes, the possible unique geometries are r-Spherical, θ-Spherical, ϕ-Spherical, $r\theta$-Spherical, $r\phi$-Spherical, $\theta\phi$-Spherical, and $r\theta\phi$-Spherical. In CÆSAR only the major geometries are included, namely x-Cartesian, r-Cylindrical, r-Spherical, xy-Cartesian, rz-Cylindrical, and xyz-Cartesian.

[1] Even though faces are two-dimensional surfaces, they may be warped in such a way that they are not flat, in the same way that the surface of a sphere is not flat even though it is two-dimensional

Mesh Type Descriptions:

In the mesh descriptions that follow, distinctions are made for the connectivity of the mesh

Unstructured Polyhedral or Polygonal Mesh

A mesh is said to be *structured*, or *logically rectangular*, if its nodes and cells can be numbered with an index for each dimension. For example, in three dimensions every cell in a structured mesh can be referred to by an (i, j, k) triplet. In a structured mesh the connectivity is known a priori – mesh cell (i, j, k) shares a face with cells $(i+1, j, k)$, $(i-1, j, k)$, $(i, j+1, k)$, $(i, j-1, k)$, $(i, j, k+1)$, and $(i, j, k-1)$ and is defined by nodes (i, j, k), $(i+1, j, k)$, $(i, j+1, k)$, $(i+1, j+1, k)$, $(i, j, k+1)$, $(i, j, k+1)$, $(i+1, j, k+1)$, $(i, j+1, k+1)$, and $(i+1, j+1, k+1)$. Structured meshes must be formed from quadrilateral meshes in 2-D and hexahedral meshes in 3-D.

A mesh is said to be *orthogonal* if all of the mesh faces are aligned with the coordinate axes. Thus, one can speak of a minus x-face or a plus y-face of a cell with no ambiguity. All angles between faces are either zero or $90°$. Orthogonal meshes are composed of rectangles in 2-D and right rectangular prisms (also known as rectangular parallelepipeds) in 3-D. In addition, each face is planar in 3-D.

A *uniform* mesh has cells that all have the same shape and size. All cells have the same volume and face areas. Stating a cell-width and the number of cells in each direction completely specifies a uniform mesh. Uniform meshes are composed of rectangles in 2-D and right rectangular prisms (also known as rectangular parallelepipeds) in 3-D.

AMR (Adaptive Mesh Refinement) meshes can be thought of as a halfway point between quadrilateral/hexahedral cell-shapes and polygonal/polyhedral cell-shapes.

Initialize (Mesh, "Uniform", Geometry, NDimensions, dx, dy, dz

Multi_Mesh public procedures:

Fundamental procedures	
`Initialize`	Initializes a Multi_Mesh object.
`Finalize`	Finalizes a Multi_Mesh object.
`Valid_State`	Returns false iff a Multi_Mesh object is in an invalid state.
`Initialized`	Returns true iff a Multi_Mesh object has been initialized.
Operations	
`Name`	Returns the name of the Multi_Mesh object.
`Output`	Writes out the Multi_Mesh object.
`Set_Version`	Sets the version number of the Multi_Mesh object (also has an assignment interface).
`Version`	Returns the version number of the Multi_Mesh object.

Multi_Mesh public defined type:

Multi_Mesh type	
`Initialized`	Initialization status.
`Name`	The name for this mesh.
`Version`	Version number which is incremented every time the array is modified, or is synced with the version number of a data structure that it depends on when it is updated.

The Multi_Mesh Class code listing in § I.1 on page 703 contains additional documentation. The Multi_Mesh Class also contains a Unit Test Program which is listed in § I.1.26 on page 775.

14.1.1 Initialize_Base_Multi_Mesh Procedure

The Initialize_Base_Multi_Mesh procedure allocates and initializes the fundamental parts of a Multi_Mesh object that are common to all mesh types. It sets the following mesh data:

- mesh type information (NDimensions, Geometry, Uniformity, Orthogonality, Structure, AMR, Shape),
- mesh scalar information (NCells_total, NCells_PE, Last_Cell_PE, First_Cell_PE, Range_Cells_PE, NNodes_total, NNodes_PE, Last_Node_PE, First_Node_PE, Range_Nodes_PE, NFaces_total, NFaces_PE, Last_Face_PE, First_Face_PE, Range_Faces_PE, Nodes_per_Cell, Nodes_per_Face, Faces_per_Cell),
- mesh base structures (Node_Structure, Cell_Structure, Face_Structure),
- mesh coordinates (Coordinates_Nodes), and
- mesh node-cell connectivity (Nodes_of_Cells_Index).

The Initialize_Base_Multi_Mesh procedure is most often used in specific mesh initialization procedures.

Calling syntax:

```
call Initialize (Mesh, NDimensions, Geometry, Uniformity, Orthogonality, Structure, AMR, Shape, NNodes_Vector, NCells_Vector, NFaces_Vector, Coordinates_Nodes_PE, Nodes_of_Cells_PE, Mesh_Name, status)
```

Input variables:

`Mesh`	The Multi_Mesh object to be initialized.
`NDimensions`	The number of spatial dimensions. [1,2,3]
`Geometry`	The geometry type. [Cartesian, Cylindrical, or Spherical]
`Uniformity`	The uniformity of the mesh. [Uniform, Nonuniform]
`Orthogonality`	The orthogonality of the mesh. [Orthogonal, Nonorthogonal]
`Structure`	The mesh structure. [Structured, Unstructured]
`AMR`	Logical which is true for Adaptive Mesh Refinement (AMR, H-type) meshes.
`Shape`	The shape of cells in the mesh. [Segmented, Triangular, Quadrilateral, Polygonal, Tetrahedral, Hexahedral, Polyhedral]
`NNodes_Vector`	A vector containing the number of mesh nodes on each PE.
`NCells_Vector`	A vector containing the number of mesh cells on each PE.
`NFaces_Vector`	A vector containing the number of mesh faces on each PE.
`Coordinates_Nodes_PE(NDimensions,Nodes_PE)`	The coordinates of the nodes on this PE.
`Nodes_of_Cells_PE(Cells_PE,Nodes_per_Cell)`	The node numbers for each cell on this PE.
`dim`$\{n\}$	The dimensions for this “array”. There must be dimensions specified up to a number one less than the Dimensionality. These are only needed in the second form of the call. [Optional]
`Mesh_Name`	The name for this mesh. [Optional]

Output variables:

`Mesh`	The Multi_Mesh object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Base_Multi_Mesh code listing in § I.1.1 on page 710 contains additional documentation.

14.1.2 Initialize_Uniform_Multi_Mesh Procedure

The Initialize_Uniform_Multi_Mesh procedure allocates and initializes a uniform Multi_Mesh object. A uniform mesh is a structured, orthogonal, cartesian[2] mesh where every cell is exactly the same size and shape. There is a single Δx, Δy and Δz for the entire mesh, which has a block-shaped domain (i.e. a right rectangular prism in 3D). In parallel, each PE also contains a block-shaped domain, but each PE may have a different-sized block. An optimum partitioning of the mesh, given these constraints, is generated by the Gen_StructureMesh_Connectivity procedure.

In addition to the mesh data set by the Initialize_Base_Multi_Mesh procedure, this procedure also sets the following uniform-specific mesh data:

- mesh cell-cell connectivity (Cells_of_Cells_Index),
- mesh face flags to indicate left (-x), right (+x), front (-y), back (+y), bottom (-z), top (+z) and interior faces (Flag_Faces_of_Cells),
- physical dimensions of the entire domain (Lengths),
- the volume of every cell (Volume_All_Cells), and
- the area for all faces (Area_All_Faces).

Calling syntax:

```
call Initialize (Mesh, NDimensions, Lengths, NCells_X_total, NCells_Y_total, NCells_Z_total, Mesh_Name, status)
```

Input variables:

`Mesh`	The Multi_Mesh object to be initialized.
`NDimensions`	The number of dimensions for the mesh.
`Lengths`	A vector of the physical lengths for the mesh.
`NCells_X_total`	Total number of cells in the X-direction, defined on every PE.
`NCells_Y_total`	Total number of cells in the Y-direction, defined on every PE. [Optional]
`NCells_Z_total`	Total number of cells in the Z-direction, defined on every PE. [Optional]
`Mesh_Name`	The name for this mesh. [Optional]

Output variables:

`Mesh`	The Multi_Mesh object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Uniform_Multi_Mesh code listing in § I.1.2 on page 714 contains additional documentation.

[2]capability for non-cartesian uniform meshes could be added, but each cell would no longer have the same volume and face areas, so those procedures would need to be modified also

14.1.3 Initialize_Orthogonal_Multi_Mesh Procedure

The Initialize_Orthogonal_Multi_Mesh procedure allocates and initializes an orthogonal Multi_Mesh object. An orthogonal mesh is a structured, cartesian[3] mesh with block-shaped cells. Each cell may have a different Δx, Δy and Δz, but the entire mesh can be described by vectors of coordinates along the three axes. The mesh has a block-shaped domain (i.e. a right rectangular prism in 3D). In parallel, each PE also contains a block-shaped domain, but each PE may have a different-sized block. An optimum partitioning of the mesh, given these constraints, is generated by the Gen_StructureMesh_Connectivity procedure.

In addition to the mesh data set by the Initialize_Base_Multi_Mesh procedure, this procedure also sets the following orthogonal-specific mesh data:

- mesh cell-cell connectivity (Cells_of_Cells_Index),
- mesh face flags to indicate left (-x), right (+x), front (-y), back (+y), bottom (-z), top (+z) and interior faces (Flag_Faces_of_Cells), and
- physical dimensions of the entire domain (Lengths).

Calling syntax:

```
call Initialize (Mesh, NDimensions, Coordinates_Nodes_X, Coordinates_Nodes_Y, Coordinates_Nodes_Z, Mesh_Name, status)
```

Input variables:

`Mesh`	The Multi_Mesh object to be initialized.
`NDimensions`	The number of dimensions for the mesh.
`Coordinates_Nodes_X`	The X-coordinates for all of the nodes, defined on every PE.
`Coordinates_Nodes_Y`	The Y-coordinates for all of the nodes, defined on every PE. [Optional]
`Coordinates_Nodes_Z`	The Z-coordinates for all of the nodes, defined on every PE. [Optional]
`Mesh_Name`	The name for this mesh. [Optional]

Output variables:

`Mesh`	The Multi_Mesh object has been allocated and initialized.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`allocate_status`	Allocation Status.
`consolidated_status`	Consolidated Status.

The Initialize_Orthogonal_Multi_Mesh code listing in § I.1.3 on page 718 contains additional documentation.

14.1.4 Finalize_Multi_Mesh Procedure

The Finalize_Multi_Mesh procedure deallocates and finalizes a Multi_Mesh object.

Calling syntax:

[3] capability for non-cartesian orthogonal meshes could be added, if the volume and face area procedures were modified accordingly

```
call Finalize (Mesh, status)
```

Input variables:

`Mesh`	The Multi_Mesh object to be finalized.

Output variables:

`Mesh`	The Multi_Mesh object has been finalized and is no longer valid.
`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

Internal variables:

`consolidated_status`	Consolidated Status.
`deallocate_status`	Deallocation Status vector.

The Finalize_Multi_Mesh code listing in § I.1.4 on page 735 contains additional documentation.

14.1.5 Valid_State_Multi_Mesh Procedure

The Valid_State_Multi_Mesh procedure returns true iff the Multi_Mesh is in a valid state – that is, iff the Multi_Mesh passes all of the valid state tests.

Calling syntax:

```
Logical = Valid_State(Mesh)
```

Input variables:

`Mesh`	The Multi_Mesh to be checked.

Output variable:

`Valid_State`	True iff the Multi_Mesh is in a valid state.

The Valid_State_Multi_Mesh code listing in § I.1.5 on page 736 contains additional documentation.

14.1.6 Initialized_Multi_Mesh Procedure

The Initialized_Multi_Mesh procedure returns true iff the Multi_Mesh object has been initialized.

Calling syntax:

```
Logical = Initialized(Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object to be examined.

Output variable:

`Initialized`	True iff the Multi_Mesh object has been initialized.

The Initialized_Multi_Mesh code listing in § I.1.6 on page 738 contains additional documentation.

14.1.7 Dump_CGNS_Multi_Mesh Procedure

NOTE: This procedure has not been fully implemented and tested.

The Dump_CGNS_Multi_Mesh procedure writes a CGNS output file for a mesh to the specified filename.

Calling syntax:

```
call Dump_CGNS (Mesh, Filename, status)
```

Input variables:

`Filename`	The filename for CGNS dump output.
`Mesh`	The Multi_Mesh object to be queried.

The Dump_CGNS_Multi_Mesh code listing in § I.1.7 on page 738 contains additional documentation.

14.1.8 Dump_GMV_Multi_Mesh Procedure

The Dump_GMV_Multi_Mesh procedure writes a GMV plotting file for a mesh and any listed variables to the specified filename.

Calling syntax:

```
call Dump_GMV (Filename, Mesh, Variable1_MV, ..., VariableN_MV, Variable1_DV, ..., VariableN_DV, status)
```

Input variables:

`Filename`	The filename for GMV dump output.
`Mesh`	The Multi_Mesh object to be output to the GMV dump file.
`Variable#_MV`	Mathematical vectors to be output along with the mesh. The pound sign may be replaced with a number from 1 to `REP_NUMBER`. [Optional]
`Variable#_DV`	Distributed vectors to be output along with the mesh. The pound sign may be replaced with a number from 1 to `REP_NUMBER`. [Optional]

Output variable:

`status`	If present, the status variable is set to either `'File Error'`, `'Memory Error'`, `'Multiple Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

The Dump_GMV_Multi_Mesh code listing in § I.1.8 on page 742 contains additional documentation.

14.1.9 Dump_GMV DV and MV Vector Procedures

The Dump_GMV_Distributed_Vector and Dump_GMV_Mathematic_Vector procedures are used internally to output a variable to the middle of a GMV plotting file.

Calling syntax:

```
call Dump_GMV_Distributed_Vector (Variable, Mesh, unit, status)
```
or
```
call Dump_GMV_Mathematic_Vector (Variable, Mesh, unit, status)
```

Input variables:

`Variable`	Mathematical or Distributed Vector to be output.
`Mesh`	The Multi_Mesh object for this GMV dump.
`unit`	The unit for GMV dump output.

Output variable:

`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

The Dump_GMV DV and MV Vector code listing in § I.1.9 on page 747 contains additional documentation.

14.1.10 Dump_XMGrace_Multi_Mesh Procedure

The Dump_XMGrace_Multi_Mesh procedure writes an XMGrace plotting file for a mesh and any listed variables to the specified filename. On a parallel run, or unless the mesh is numbered very specifically, the points will be output in a somewhat random order. These can be plotted with a scatter plot, or the numbers can be sorted by an editor or a Unix sort command.

Calling syntax:

```
call Dump_XMGrace (Filename, Mesh, Coordinate, Xmin, Xmax, Ymin, Ymax, Zmin, Zmax, Variable1_MV, ..., VariableN_MV, Variable1_DV, ..., VariableN_DV, status)
```

Input variables:

`Filename`	The filename for XMGrace dump output.
`Mesh`	The Multi_Mesh object to be output to the XMGrace dump file.
`Coordinate`	A single-character variable expressing which axis to output, either "X", "Y", or "Z".
`Xmin, Xmax`	Limits of the output variable set in the X-direction. [Optional]
`Ymin, Ymax`	Limits of the output variable set in the Y-direction. [Optional]
`Zmin, Zmax`	Limits of the output variable set in the Z-direction. [Optional]
`Variable#_MV`	Mathematical vectors to be output along with the mesh. The pound sign may be replaced with a number from 1 to `REP_NUMBER`. [Optional]
`Variable#_DV`	Distributed vectors to be output along with the mesh. The pound sign may be replaced with a number from 1 to `REP_NUMBER`. [Optional]

Output variable:

`status`	If present, the status variable is set to either `'File Error'`, `'Memory Error'`, `'Multiple Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

The Dump_XMGrace_Multi_Mesh code listing in § I.1.10 on page 749 contains additional documentation.

14.1.11 Dump_XMGrace DV and MV Vector Procedures

The Dump_XMGrace_Distributed_Vector and Dump_XMGrace_Mathematic_Vector procedures are used internally to output a variable to an XMGrace plotting file.

Calling syntax:

```
call Dump_XMGrace_Distributed_Vector (Variable, Mesh, Coordinate, unit, Xmin, Xmax, Ymin, Ymax, Zmin, Zmax, status)
```
or
```
call Dump_XMGrace_Mathematic_Vector (Variable, Mesh, Coordinate, unit, Xmin, Xmax, Ymin, Ymax, Zmin, Zmax, status)
```

Input variables:

`Variable`	Mathematical or Distributed Vector to be output.
`Mesh`	The Multi_Mesh object for this XMGrace dump.
`Coordinate`	A single-character variable expressing which axis to output, either "X", "Y", or "Z".
`Xmin, Xmax`	Limits of the output variable set in the X-direction.
`Ymin, Ymax`	Limits of the output variable set in the Y-direction.
`Zmin, Zmax`	Limits of the output variable set in the Z-direction.
`unit`	The unit for XMGrace dump output.

Output variable:

`status`	If present, the status variable is set to either `'Memory Error'` or `'Success'` depending on program execution. If not present, the procedure aborts if unsuccessful when the `DEBUG_LEVEL` is set high enough.

The Dump_XMGrace DV and MV Vector code listing in § I.1.11 on page 752 contains additional documentation.

14.1.12 Get_Area_Faces_of_Cells_Multi_Mesh Procedure

The Get_Area_Faces_of_Cells_Multi_Mesh procedure returns the vector areas for each face of each cell of a Multi_Mesh object.

Calling syntax:

```
call Get_Area_Faces_of_Cells (Area_Faces_of_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`Area_Faces_of_Cells`	An array of the vector areas for each face of the cells on this PE.

The Get_Area_Faces_of_Cells_Multi_Mesh code listing in § I.1.12 on page 755 contains additional documentation.

14.1.13 Get_Coordinates_Cells_Multi_Mesh Procedure

The Get_Coordinates_Cells_Multi_Mesh procedure returns the cell center coordinates for each cell of a Multi_Mesh object.

Calling syntax:

```
call Get_Coordinates_Cells (Coordinates_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`Coordinates_Cells`	The cell-center coordinates for each cell on this PE.

The Get_Coordinates_Cells_Multi_Mesh code listing in § I.1.13 on page 757 contains additional documentation.

14.1.14 Get_Coordinates_Cells_of_Cells_Multi_Mesh Procedure

The Get_Coordinates_Cells_of_Cells_Multi_Mesh procedure returns the coordinates for the cell on the other side of each face of each cell of a Multi_Mesh object.

Calling syntax:

```
call Get_Coordinates_Cells_of_Cells (Coordinates_Cells_of_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`Coordinates_Cells_of_Cells`	An array of the coordinates for the cells on the other side of each face of the cells on this PE.

The Get_Coordinates_Cells_of_Cells_Multi_Mesh code listing in § I.1.14 on page 758 contains additional documentation.

14.1.15 Get_Coordinates_Faces_of_Cells_Multi_Mesh Procedure

The Get_Coordinates_Faces_of_Cells_Multi_Mesh procedure returns the coordinates for each face of each cell of a Multi_Mesh object.

Calling syntax:

```
call Get_Coordinates_Faces_of_Cells (Coordinates_Faces_of_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`Coordinates_Faces_of_Cells`	An array of the coordinates for each face of the cells on this PE.

The Get_Coordinates_Faces_of_Cells_Multi_Mesh code listing in § I.1.15 on page 761 contains additional documentation.

14.1.16 Get_Coordinates_Nodes_of_Cells_Multi_Mesh Procedure

The Get_Coordinates_Nodes_of_Cells_Multi_Mesh procedure returns the node coordinates for each node of each cell of a Multi_Mesh object.

Calling syntax:

```
call Get_Coordinates_Nodes_of_Cells (Coordinates_Nodes_of_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`Coordinates_Nodes_of_Cells`	The coordinates of the nodes for each cell on this PE.

The Get_Coordinates_Nodes_of_Cells_Multi_Mesh code listing in § I.1.16 on page 763 contains additional documentation.

14.1.17 Get_DeltaR21_Cells_of_Cells_Multi_Mesh Procedure

The Get_DeltaR21_Cells_of_Cells_Multi_Mesh procedure returns the absolute distance between the cell center and the cell center across each face (the "other" cell) for each face of each cell of a Multi_Mesh object.

Calling syntax:

```
call Get_DeltaR21_Cells_of_Cells (DeltaR21_Cells_of_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`DeltaR21_Cells_of_Cells`	The absolute distance between the cell center and the other_cell center (across the face) for each cell on this PE.

The Get_DeltaR21_Cells_of_Cells_Multi_Mesh code listing in § I.1.17 on page 764 contains additional documentation.

14.1.18 Get_DeltaR1f_Cells_of_Cells_Multi_Mesh Procedure

The Get_DeltaR1f_Cells_of_Cells_Multi_Mesh procedure returns the absolute distance between the cell center and each face center for each cell of a Multi_Mesh object. Although the locus could be thought of as

Faces_of_Cells, it is called Cells_of_Cells to correspond with other variables across the face (e.g. DeltaR2f_-Cells_of_Cells).

Calling syntax:

```
call Get_DeltaR1f_Cells_of_Cells (DeltaR1f_Cells_of_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`DeltaR1f_Cells_of_Cells`	The absolute distance between the cell center and each face center for each cell on this PE.

The Get_DeltaR1f_Cells_of_Cells_Multi_Mesh code listing in § I.1.18 on page 765 contains additional documentation.

14.1.19 Get_DeltaR2f_Cells_of_Cells_Multi_Mesh Procedure

The Get_DeltaR2f_Cells_of_Cells_Multi_Mesh procedure returns the absolute distance between the cell center across the face (the "other" cell) and the face itself for each face of each cell of a Multi_Mesh object.

Calling syntax:

```
call Get_DeltaR2f_Cells_of_Cells (DeltaR2f_Cells_of_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`DeltaR2f_Cells_of_Cells`	The absolute distance between the other cell center and each face center for each cell on this PE.

The Get_DeltaR2f_Cells_of_Cells_Multi_Mesh code listing in § I.1.19 on page 766 contains additional documentation.

14.1.20 Get_Flag_Faces_of_Cells_Multi_Mesh Procedure

The Get_Flag_Faces_of_Cells_Multi_Mesh procedure returns the Flag for each face of each cell of a Multi_Mesh object. These flags are set to:

0	internal faces
1	left faces (-x)
2	right faces (+x)
3	front faces (-y)
4	back faces (+y)
5	bottom faces (-z)
6	top faces (+z)

Calling syntax:

```
call Get_Flag_Faces_of_Cells (Flag_Faces_of_Cells, Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object.

Output variable:

`Flag_Faces_of_Cells`	The flags for each face of the cells on this PE.

The Get_Flag_Faces_of_Cells_Multi_Mesh code listing in § I.1.20 on page 768 contains additional documentation.

14.1.21 Get Value Multi_Mesh Functions

The Get_Value_Multi_Mesh functions return values from a Multi_Mesh object.

Calling syntax:

```
Pointer = Cell_Structure(Mesh)      ,
Pointer = Face_Structure(Mesh)      ,
Output = First_Cell_PE(Mesh)        ,
Output = First_Face_PE(Mesh)        ,
Output = First_Node_PE(Mesh)        ,
Output = Get_Faces_per_Cell(Mesh)   ,
Output = Get_NDimensions(Mesh)      ,
Output = Last_Cell_PE(Mesh)         ,
Output = Last_Face_PE(Mesh)         ,
Output = Last_Node_PE(Mesh)         ,
Output = Name(Mesh)                 ,
Output = NCells_PE(Mesh)            ,
Output = NCells_Total(Mesh)         ,
Output = NFaces_PE(Mesh)            ,
Output = NFaces_Total(Mesh)         ,
Output = NNodes_PE(Mesh)            ,
Output = NNodes_Total(Mesh)         ,
Pointer = Node_Structure(Mesh)      ,
Range = Range_Cells_PE(Mesh)        ,
Range = Range_Faces_PE(Mesh)        or
Range = Range_Nodes_PE(Mesh)
```

Input variable:

`Mesh`	The Multi_Mesh object to be queried.

Output variables:

`Pointer`	A pointer to a Base Structure within the Mesh object.
`Output`	For Name, returns a character variable containing the name assigned to the Mesh upon initialization. Otherwise, an integer with the named value from the Mesh object is returned.
`Range`	A dimension(2) integer with the specified range from the Mesh object.

The Get Value Multi_Mesh code listing in § I.1.21 on page 769 contains additional documentation.

14.1.22 Get_Version_Multi_Mesh Procedure

The Get_Version_Multi_Mesh procedure returns the version number for a Multi_Mesh Object.

Calling syntax:

```
Integer = Version(Mesh)
```

Input variables:

`Mesh` The Multi_Mesh object to be queried.

Output variable:

`Version` The version number of the Multi_Mesh object.

The Get_Version_Multi_Mesh code listing in § I.1.22 on page 771 contains additional documentation.

14.1.23 Get_Volume_Cells_Multi_Mesh Procedure

The Get_Volume_Cells_Multi_Mesh procedure returns the cell volumes from a Multi_Mesh object.

Calling syntax:

```
call Get_Volume_Cells (Volume_Cells, Mesh)
```

Input variable:

`Mesh` The Multi_Mesh object.

Output variable:

`Volume_Cells` An array of the cell volumes on this PE.

The Get_Volume_Cells_Multi_Mesh code listing in § I.1.23 on page 772 contains additional documentation.

14.1.24 Set_Coordinates_Multi_Mesh Procedure

Note: This procedure is not finished and does not currently work.

The Set_Coordinates_Multi_Mesh procedure sets the coordinates for the Multi_Mesh object.

Calling syntax:

```
Mesh = Coordinates                          or
call Set_Coordinates (Mesh, Coordinates)
```

Input variable:

`Coordinates` The Bare Naked Array of coordinates for the Multi_Mesh object, defined differently on each PE.

Input/Output variable:

`Mesh` The Multi_Mesh object to be set.

Internal variable:

`Version_Increment` The amount that the version number is incremented, which is a global class variable.

The Set_Coordinates_Multi_Mesh code listing in § I.1.24 on page 773 contains additional documentation.

14.1.25 Set_Version_Multi_Mesh Procedure

The Set_Version_Multi_Mesh procedure sets the version number for the Multi_Mesh object.

Calling syntax:

```
Mesh = Version                         or
call Set_Version (Mesh, Version)
```

Input variable:

`Version` The version number for the Multi_Mesh object.

Input/Output variable:

`Mesh` The Multi_Mesh object to be set.

The Set_Version_Multi_Mesh code listing in § I.1.25 on page 774 contains additional documentation.

Part III

Cæsar Package Methods Discussion

Chapter 15

Mathematics Methods

This chapter describes the general mathematical methods used in CÆSAR. Each section uses its own unique nomenclature.

15.1 Math_Utils Methods

The CÆSAR Math_Utils Module contains the following procedures:

Math_Utils public procedures:

`Prime_Factors`	Returns a vector containing the prime factorization of a number.

15.1.1 Prime_Factors Procedure

The Prime_Factors procedure uses an optimized direct search procedure to calculate the prime factorization of a number N:

- Factors of N are found by systematically performing trial divisions using a sequence of increasing numbers.
- Note that a direct search to discover the prime factors does not need to go further than $\sqrt{N}$, since all integers greater than $\sqrt{N}$ have already had their cofactors tested.
- Recognize that if, for example, all factors of 2 are factored out of a number (to yield $N' = N/2^p$), then the problem has been reduced to finding the prime factorization of the smaller number N', using integers between 3 and $\sqrt{N'}$.
- Also, a procedure is used to eliminate multiples of small primes from the testing procedure. Once the first two primes have been factored out of the number (to yield $N'' = N/\left(2^p 3^q\right)$), then the remaining primes are included within the set $\{6i-1, 6i+1\}$. This is true because the entire set of integers can be represented as $\{6i, 6i+1, 6i+2, 6i+3, 6i+4, 6i+5\}$; all but $\{6i+1, 6i+5\}$ are divisible by 2 and/or 3; and $\{6i+1, 6i+5\}$ is equivalent to $\{6i-1, 6i+1\}$. Therefore, only integers of the forms $6i-1$ and $6i+1$ need be checked.

15.2 Statistics Methods

The CÆSAR Statistics Class is used to keep track of the statistics associated with a sample population or distribution. Currently, the class stores only conglomerate information about the population and does not store the entire distribution. Therefore, certain types of statistical information, such as the median and the mode, are not available. The following values are calculated by the class.

Given a set of N variables denoted by $\{x_i\}$, the arithmetic mean, or average, of the distribution is given by

$$\langle x \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i \ . \tag{15.1}$$

The geometric mean of the distribution (calculated only if all x_i are positive) is given by

$$\langle x \rangle_G = \sqrt[N]{\prod_{i=1}^{N} x_i} \ , \tag{15.2}$$

or equivalently by

$$\langle x \rangle_G = \exp\left(\frac{1}{N} \sum_{i=1}^{N} \ln x_i \right) \ . \tag{15.3}$$

The harmonic mean of the distribution is given by:

$$\langle x \rangle_H = \left[\frac{1}{N} \sum_{i=1}^{N} \frac{1}{x_i} \right]^{-1} \ . \tag{15.4}$$

The standard deviation of the distribution is given by

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} \left(x_i - \langle x \rangle\right)^2} \ , \tag{15.5}$$

or equivalently by

$$s = \sqrt{\frac{1}{N-1} \left[\sum_{i=1}^{N} (x_i^2) - N \langle x \rangle^2 \right]} \ . \tag{15.6}$$

Note that the factor $N-1$, rather than N, is required in the denominator to account for the fact that the parameter $\langle x \rangle$ has been determined from the distribution and not independently. This formula for the standard deviation is sometimes called the sample standard deviation. The limits of the sample mean and the sample standard deviation give the true values:

$$\mu = \lim_{N \to \infty} \langle x \rangle \ , \tag{15.7}$$
$$\sigma = \lim_{N \to \infty} s \ . \tag{15.8}$$
$$\tag{15.9}$$

The Statistics Class also calculates minimum and maximum values for the distribution. The Valid_State procedure verifies that all of the means lie within the extremum bounds, and that the following mathematical relationship holds:

$$\langle x \rangle_H \le \langle x \rangle_G \le \langle x \rangle \ . \tag{15.10}$$

Chapter 16

Linear Algebra Methods

This chapter describes the general linear algebra methods used in CÆSAR. Each section uses its own unique nomenclature.

16.1 Mathematic_Vector Methods

The CÆSAR Code Package uses the Mathematic_Vector class to contain and manipulate algebraic vectors. The following discussion assumes that x and y are vectors of length N.

The CÆSAR Code Package contains procedures to calculate norms and dot products of vectors. In parallel, these operations require global communication. Dot products are defined by

$$x^T y = x \cdot y = (x, y) = \sum_{i=1,N} x_i \, y_i \, . \tag{16.1}$$

A vector norm is any scalar-valued function on a vector, denoted by the double bar notation $\|x\|$, that satisfies these properties:

$$\begin{array}{ll} \|x\| \geq 0 & (\|x\| = 0 \text{ iff } x = 0) \, , \\ \|x + y\| \leq \|x\| + \|y\| \, , & \\ \|sx\| = |s| \, \|x\| \, , & \text{where } s \text{ is a scalar} \, . \end{array} \tag{16.2}$$

The general class of vector norms known as the p-norms are defined by

$$\|x\|_p = \sqrt[p]{\sum_{i=1,N} |x_i|^p} \qquad , \, p \geq 1 \, . \tag{16.3}$$

In particular, the 1, 2, and ∞ norms are the most important:

$$\|x\|_1 = \sum_{i=1,N} |x_i| \, , \tag{16.4}$$

$$\|x\|_2 = \sqrt{\sum_{i=1,N} |x_i|^2} = \sqrt{x^T x} \, , \qquad \text{and} \tag{16.5}$$

$$\|x\|_\infty = \max_{1 \leq i \leq N} |x_i| \, . \tag{16.6}$$

The following vector norm relationships are verified by the Mathematic_Vector Valid_State procedure:

$$\begin{array}{ccccc} \|x\|_2 & \leq & \|x\|_1 & \leq & \sqrt{N} \, \|x\|_2 \, , \\ \|x\|_\infty & \leq & \|x\|_2 & \leq & \sqrt{N} \, \|x\|_\infty \, , \\ \|x\|_\infty & \leq & \|x\|_1 & \leq & N \, \|x\|_\infty \, . \end{array} \tag{16.7}$$

The Cauchy-Schwartz inequality is verified by the Mathematic_Vector DotProduct procedure:

$$\left|x^T y\right| \le \|x\|_2 \, \|y\|_2 \ . \tag{16.8}$$

For a similar but more complete discussion of vector operations see Golub and Van Loan (1989).

16.2 ELL_Matrix Methods

The CÆSAR Code Package uses the ELL_Matrix class to contain and manipulate algebraic matrices. The following discussion assumes that A and B are matrices of dimension $M \times N$, $a_{i,j}$ is an element of A, x is a vector of length N, and y is a vector of length M.

The CÆSAR Code Package contains procedures to calculate matrix norms and matrix-vector products. In parallel, these operations require global communication. Matrix-vector products (MatVecs) are defined by

$$Ax = y = \sum_{j=1,N} a_{i,j} \, x_j \ . \tag{16.9}$$

A matrix norm is any scalar-valued function on a matrix, denoted by the double bar notation $\|A\|$, that satisfies these properties:

$$\begin{array}{ll} \|A\| \ge 0 & (\|A\| = 0 \text{ iff } A = 0) \ , \\ \|A + B\| \le \|A\| + \|B\| \ , & \\ \|sA\| = |s| \, \|A\| \ , & \text{where } s \text{ is a scalar} \ . \end{array} \tag{16.10}$$

A frequently used matrix norm is the Frobenius norm:

$$\|A\|_F = \sqrt{\sum_{i=1,M} \sum_{j=1,N} |a_{i,j}|^2} \ . \tag{16.11}$$

The general class of matrix norms known as the p-norms are defined in terms of vector norms by:

$$\|A\|_p = \sup_{x \ne 0} \frac{\|Ax\|_p}{\|x\|_p} = \max_{x:\|x\|_p = 1} \|Ax\|_p \qquad , \ p \ge 1 \ . \tag{16.12}$$

In particular, the 1 and ∞ norms are the most easily calculated:

$$\|A\|_1 = \max_{1 \le j \le N} \sum_{i=1,M} |a_{i,j}| \qquad \text{(maximum absolute column sum),} \tag{16.13}$$

$$\|A\|_\infty = \max_{1 \le i \le M} \sum_{j=1,N} |a_{i,j}| \qquad \text{(maximum absolute row sum).} \tag{16.14}$$

In general, p-norms are difficult to calculate. The 2-norm can be shown to be:

$$\|A\|_2 = \sqrt{\text{maximum eigenvalue of } A^H A} \ , \tag{16.15}$$

$$= \sqrt{\text{maximum eigenvalue of } A^T A} \ , \qquad \text{if } A \text{ is real} \ , \tag{16.16}$$

where A^H is the Hermitian (or complex conjugate transpose, or adjoint) of A. Calculating the 2-norm of a matrix requires an iterative procedure, and is not currently done in CÆSAR (only Frobenius, $p = 1$ and $p = \infty$ norms are calculated). However, CÆSAR calculates an estimate of the 2-norm as a range (or the middle of the range) using the following relationships:

$$\begin{array}{rcccl} \frac{1}{\sqrt{N}} \|A\|_F & \le & \|A\|_2 & \le & \|A\|_F \ , \\ \max\limits_{i,j} |a_{i,j}| & \le & \|A\|_2 & \le & \sqrt{MN} \max\limits_{i,j} |a_{i,j}| \ , \\ \frac{1}{\sqrt{N}} \|A\|_\infty & \le & \|A\|_2 & \le & \sqrt{M} \, \|A\|_\infty \ , \\ \frac{1}{\sqrt{M}} \|A\|_1 & \le & \|A\|_2 & \le & \sqrt{N} \, \|A\|_1 \ , \\ & & \|A\|_2 & \le & \sqrt{\|A\|_1 \, \|A\|_\infty} \ . \end{array} \tag{16.17}$$

For a similar but more complete discussion of matrix operations see Golub and Van Loan (1989).

16.3 Solver Methods

The CÆSAR Code Package uses the blorp method to calculate Solvers.

Part IV

Cæsar Package Code Listings

Appendix A

m4 Preprocessing Code Listings

A.1 Settings m4 Macros

The main documentation of the Settings m4 Macros in section 6.1 contains additional explanation of this code listing.

```
dnl
dnl Author: Michael L. Hall
dnl         P.O. Box 1663, MS-D409, LANL
dnl         Los Alamos, NM 87545
dnl         ph: 505-665-4312
dnl         email: hall@lanl.gov
dnl
dnl Created on: 02/26/98
dnl Date:       03/21/00, 17:15:20
dnl Version:    6.8


dnl Change the quotation characters to avoid F90 conflicts.

m4_changequote([,])


dnl Change the comment characters to eliminate m4 macro
dnl expansion in F90 comments.

m4_changecom([!])


dnl Redefine the "dnl" command to be unprefixed. Note that the "dnl"
dnl command is nonfunctional before this definition is made, but that
dnl unwanted output from the processing of this file is avoided by
dnl diverting the entire output (up until now) to nowhere. (Notice the
dnl m4_divert(-1) command in the initialization.)

m4_define([dnl],m4_defn([m4_dnl]))
m4_divert
```

```
dnl Define unprefixed versions of m4 builtin macros that are
dnl commonly used. All other m4 builtin macros must be prefixed
dnl by "m4_".

m4_define([define],m4_defn([m4_define]))
m4_define([ifdef],m4_defn([m4_ifdef]))
m4_define([ifelse],m4_defn([m4_ifelse]))
m4_define([include],m4_defn([m4_include]))
m4_define([popdef],m4_defn([m4_popdef]))
m4_define([pushdef],m4_defn([m4_pushdef]))
m4_define([undefine],m4_defn([m4_undefine]))


dnl The m4_chop macro (based on perl's chop command) returns the
dnl input string minus its final character. m4_chop is useful for
dnl removing "\n" from m4_esyscmd strings.

define([m4_chop], [m4_substr($1, 0, m4_decr(m4_len($1)))])


dnl Define cpp-style predefined macros for date and time information.
dnl These names correspond to those described in "The C Programming
dnl Language", B. W. Kernighan and D. M. Ritchie (1988), page 233.
dnl The __file__ and __line__ predefined macros are already defined
dnl in Gnu m4, but are redefined here to remove the "m4_" prefix.

define([__date__], [m4_chop(m4_esyscmd(date +%D))])
define([__time__], [m4_chop(m4_esyscmd(date +%T))])
define([__file__], m4_defn([m4___file__]))
define([__line__], m4_defn([m4___line__]))


dnl The m4_die macro (based on perl's die command) prints an error
dnl message and then terminates.

define([m4_die],[m4_errprint([m4:]__FILE__:__LINE__: [$1]) m4_m4exit(1)])


dnl The expand macro is used to force macro expansion in a word
dnl containing underscores, since underscores are not a word-delimiter
dnl in standard m4.

define([expand], [m4_translit(m4_translit([$1], [_], [*]), [*], [_])])


dnl The forloop macro, which iterates a section of text numerically,
dnl letting the loop variable take on successive numerical values when
dnl expressing the section of text. This version is better than the
dnl version in the Gnu m4 manual because it checks for the case of
dnl starting loop value greater than final loop value, and produces
dnl no output when this occurs.
dnl
dnl The arguments for the forloop macro are:
```

```
dnl
dnl   $1 - The name of the iteration variable.
dnl   $2 - The starting value.
dnl   $3 - The final value.
dnl   $4 - The text to be expanded for each iteration.

define([forloop],
[ifelse(
  m4_eval($2 > $3),  1,
    [],
  m4_eval($2 <= $3), 1,
    [pushdef([$1], [$2])$4[]popdef([$1])])[]dnl
ifelse(
  m4_eval($2 < $3), 1,
    [forloop([$1], m4_incr($2), [$3], [$4])])])


dnl The firstword and tailwords macros operate on a space-delimited list
dnl of words. The firstword macro returns the first word in the list, and
dnl the tailwords macro returns a space-delimited list of words derived
dnl by eliminating the first word.

define([firstword],
[ifelse(
  m4_index([$1], [ ]), [-1],
    [$1],
    [m4_substr([$1],0,m4_index([$1], [ ]))])[]dnl
])

define([tailwords],
[ifelse(
  m4_index([$1], [ ]), [-1],
    [],
    [m4_substr([$1],m4_incr(m4_index([$1], [ ])))])[]dnl
])


dnl The fortext macro is similar to the forloop macro, except that the
dnl loop variable takes on successive textual values instead of numerical
dnl values. The arguments for the fortext macro are:
dnl
dnl   $1 - The name of the iteration variable.
dnl   $2 - A string of values for the iteration variable, separated by
dnl        spaces.
dnl   $3 - The text to be expanded for each iteration.

define([fortext],
[ifelse(
  firstword($2), [],
    [],
    [pushdef([$1],[firstword($2)])$3[]fortext(
      [$1], tailwords($2), [$3])])popdef([$1])[]dnl
])
```

A.2 Type m4 Macros

The main documentation of the Type m4 Macros in section 6.2 contains additional explanation of this code listing.

```
dnl
dnl Author: Michael L. Hall
dnl         P.O. Box 1663, MS-D409, LANL
dnl         Los Alamos, NM 87545
dnl         ph: 505-665-4312
dnl         email: hall@lanl.gov
dnl
dnl Created on: 11/12/98
dnl Date:       03/21/00, 17:16:07
dnl Version:    4.0


dnl Define a real$kind private macro based on the desired precision.

ifdef([SINGLE],[
  define([real$kind],[1.0e0])
],[
  ifdef([UNICOS],[
    define([real$kind],[1.0e0])
  ],[
    define([real$kind],[1.0d0])
  ])
])


dnl Define a private macro to expand to the pointer and dimensioning
dnl info.

define([pnt$dim],
       [ifelse(
         [$1], [],
           [],
         [$1], [0],
           [],
         [ifelse([$2], [], [, pointer]), dimension(:forloop([i],2,$1,[,:]))])])


dnl Define a type macro which interprets real, integer, logical
dnl and character types correctly and leaves other types unchanged.
dnl
dnl Arguments to the type macro:
dnl   $1 - The main type: real, integer, logical or character. If the
dnl        type is character, the next argument is the character
dnl        length (and all the remaining argument numbers are
dnl        incremented). If another word appears as argument 1 ($1),
dnl        then no action is taken, which is correct for a derived
```

```
dnl        type.
dnl   $2 - The rank of the variable, between zero and seven (default =
dnl        zero).
dnl   $3 - If anything appears here, then the variable is *not* declared
dnl        to be a pointer, but otherwise it is. An exception is that
dnl        scalars are never declared to be pointers.

define([type],
       [ifelse(
         [$1], [real],
           [real (kind=KIND(m4_indir(real$kind)))m4_indir(pnt$dim,$2,$3)],
         [$1], [integer],
           [integer (kind=KIND(1))m4_indir(pnt$dim,$2,$3)],
         [$1], [logical],
           [logical (kind=KIND(.true.))m4_indir(pnt$dim,$2,$3)],
         [$1], [character],
           [character (len=$2)m4_indir(pnt$dim,$3,$4)],
         [$1], [],
           [[type]],
         [[type]($*)])])


dnl Define a changetype macro which interprets real and integer type
dnl conversions correctly and emits an error message if incorrectly
dnl called.

define([changetype],
       [ifelse(
         [$1], [real],
           [REAL($2, KIND(m4_indir(real$kind)))],
         [$1], [integer],
           [INT($2, KIND(1))],
         [m4_die([Error in changetype command.])])])
```

A.3 Verify m4 Macros

The main documentation of the Verify m4 Macros in section 6.3 contains additional explanation of this code listing.

```
dnl
dnl Author: Michael L. Hall
dnl         P.O. Box 1663, MS-D409, LANL
dnl         Los Alamos, NM 87545
dnl         ph: 505-665-4312
dnl         email: hall@lanl.gov
dnl
dnl Created on: 02/19/98
dnl Date:       09/18/00, 20:45:35
dnl Version:    6.0


dnl Initialize the DEBUG_LEVEL and WARNING_LEVEL to zero if they
```

```
dnl are not already defined.

ifdef([DEBUG_LEVEL], [
],[
  define([DEBUG_LEVEL], 0)
])

ifdef([WARNING_LEVEL], [
],[
  define([WARNING_LEVEL], 0)
])


dnl Set the default communication style.

define([VERIFY_COMMUNICATION],[Global])


dnl Define the VERIFY macro.

define([VERIFY], [
  ifelse(m4_eval(DEBUG_LEVEL >= $2), 1, [
    define([HIDE], [])
  ],m4_eval(DEBUG_LEVEL == -1), 1, [
    define([HIDE], [[!]])
  ],[
    define([HIDE], [[!]])
  ])
  define([COMMAND_TEXT],m4_changequote(["],["])$1[]m4_changequote("[","]"))
  ifelse(VERIFY_COMMUNICATION, Global, [
    HIDE if (.not. Global_ALL($1)) then
    HIDE   if (this_is_IO_PE) then
    HIDE     write (6,*) "Verification failed: ", &
    HIDE                 "COMMAND_TEXT, ", &
    HIDE                 "file __file__, ", &
    HIDE                 "line __line__."
    HIDE   end if
    HIDE   call Abort
    HIDE end if
  ],[
    HIDE if (.not. ($1)) then
    HIDE   write (6,*) "Verification failed: ", &
    HIDE               "COMMAND_TEXT, ", &
    HIDE               "file __file__, ", &
    HIDE               "line __line__."
    HIDE   stop
    HIDE end if
  ])
  undefine([COMMAND_TEXT])
  undefine([HIDE])
])


dnl Define the WARN_IF macro.
```

```
define([WARN_IF], [
  ifelse(m4_eval(WARNING_LEVEL >= $2), 1, [
    define([HIDE], [])
  ],m4_eval(WARNING_LEVEL == -1), 1, [
    define([HIDE], [[!]])
  ],[
    define([HIDE], [[!]])
  ])
  define([COMMAND_TEXT],m4_changequote(["],["])$1[]m4_changequote("[","]"))
  ifelse(VERIFY_COMMUNICATION, Global, [
    HIDE if (Global_ANY($1)) then
    HIDE   if (this_is_IO_PE) then
    HIDE     write (6,*) "Warning - test failed: ", &
    HIDE                 "COMMAND_TEXT, ", &
    HIDE                 "file __file__, ", &
    HIDE                 "line __line__."
    HIDE   end if
    HIDE end if
  ],[
    HIDE if ($1) then
    HIDE   write (6,*) "Warning - test failed: ", &
    HIDE               "COMMAND_TEXT, ", &
    HIDE               "file __file__, ", &
    HIDE               "line __line__."
    HIDE end if
  ])
  undefine([COMMAND_TEXT])
  undefine([HIDE])
])
```

A.4 Replicate m4 Macros

The main documentation of the Replicate m4 Macros in section 6.4 contains additional explanation of this code listing.

```
dnl
dnl Author: Michael L. Hall
dnl         P.O. Box 1663, MS-D409, LANL
dnl         Los Alamos, NM 87545
dnl         ph: 505-665-4312
dnl         email: hall@lanl.gov
dnl
dnl Created on: 12/04/98
dnl Date:       03/21/00, 17:18:38
dnl Version:    4.9


dnl Define the REPLICATE macro.

define([REPLICATE],[
```

```
  forloop([i], 0, 7, [define([REP_NUMBER],[i]) REPLICATE_ROUTINE(i)])
])


dnl Define the REPLICATE_ARRAYS macro.

define([REPLICATE_ARRAYS],[
  forloop([i], 1, 7, [define([REP_NUMBER],[i]) REPLICATE_ROUTINE(i)])
])


dnl Define the ARRAY_ONLY macro.

define([ARRAY_ONLY],
[ifelse(
  [0], REP_NUMBER,
    [!],
  [])])


dnl Define the SCALAR_ONLY macro.

define([SCALAR_ONLY],
[ifelse(
  [0], REP_NUMBER,
    [],
  [!])])


dnl Define the REP_EXPAND command, which expands text with a glorified do-loop.
dnl The arguments are:
dnl
dnl   REP_NUMBER - The number of iterations.
dnl   $1 - Beginning text that is present if the iteration count is non-zero.
dnl   $2 - The text to be repeated in each iteration. The iteration variable,
dnl        i, may be used in this text.
dnl   $3 - Separator text to be put between iterations.
dnl   $4 - Final text that is present if the iteration count is non-zero.
dnl
dnl So, an iteration count of "0" yields no output, and an iteration count
dnl of "4" yields:
dnl
dnl   $1$2$3$2$3$2$3$2$4
dnl
dnl Note that the separator text ($3) appears only 3 times.

define([REP_EXPAND],
[ifelse([0], REP_NUMBER,
  [],
  [$1[]forloop([i], 1, 1, [$2])])dnl
ifelse(m4_eval(REP_NUMBER >= 2), 1,
  [forloop([i], 2, REP_NUMBER, [$3][$2])])dnl
ifelse([0], REP_NUMBER,
  [],
```

```
  [$4])])


dnl Define the REPLICATE_INTERFACE macro.

define([REPLICATE_INTERFACE],[
  interface $1
forloop([i], 0, 7, [    module procedure $2_[]i
])dnl
  end interface
])


dnl Define some useful macros based on the REP_EXPAND macro.

dnl REP_DECLARE:    form -->  real :: var1, var2, var3
dnl
dnl  $1 - Declaration type (everything up to the double colon).
dnl  $2 - Variable name to replicate, which should contain an "i"
dnl       for the iteration number.

define([REP_DECLARE],
  [REP_EXPAND([$1 :: ], [$2], [, ], [])])

dnl REP_ALLOCATE:   form -->  ALLOCATE(R(var1, var2, var3))
dnl
dnl  $1 - Variable to allocate.
dnl  $2 - Dimensioning variable name to replicate, which should
dnl       contain an "i" for the iteration number.
dnl  $3 - Name for the status variable.

define([REP_ALLOCATE],
  [REP_EXPAND([ALLOCATE($1(], [$2], [, ], [), stat=$3)])])

dnl REP_ARGS:     form -->  , var1, var2, var3
dnl
dnl  $1 - Variable name to replicate, which should contain an "i"
dnl       for the iteration number.

define([REP_ARGS],
  [REP_EXPAND([, ], [$1], [, ], [])])


dnl Input text used to generate documentation:

dnl module test_replicate
dnl   REPLICATE_INTERFACE([Generic_Routine], [Specific_Routine])
dnl contains
dnl
dnl define([REPLICATE_ROUTINE],[
dnl   function Specific_Routine_$1 (R[]REP_ARGS([var[]i]))
dnl     type(real,$1) :: R
dnl     REP_DECLARE([type(integer)], [var[]i])
dnl     REP_ALLOCATE(R, [var[]i], [status])
```

```
dnl     ARRAY_ONLY DEALLOCATE(R)
dnl     SCALAR_ONLY R = 999.
dnl     <other routine contents>
dnl   end function Specific_Routine_$1
dnl
dnl ])
dnl REPLICATE
dnl end module test_replicate
```

A.5 Superclass m4 Macros

The main documentation of the Superclass m4 Macros in section 6.5 contains additional explanation of this code listing.

```
dnl
dnl Author: Michael L. Hall
dnl         P.O. Box 1663, MS-D409, LANL
dnl         Los Alamos, NM 87545
dnl         ph: 505-665-4312
dnl         email: hall@lanl.gov
dnl
dnl Created on: 1/20/99
dnl Date:       03/21/00, 17:19:14
dnl Version:    4.6


dnl The MAKE_INTERFACES macro expands into standard interface
dnl specifications using the arguments:
dnl
dnl  $1 - Specific Class Suffix
dnl  $2 - Generic Interface Names (separated by spaces)

define([MAKE_INTERFACES],[
  public :: m4_patsubst(m4_shift($@), [ ], [, ])

fortext([BASENAME], m4_shift($@), [
  interface BASENAME
    module procedure BASENAME[]_[]$1
  end interface
]) ])
dnl MAKE_INTERFACES([One], [Initialize Verify_State Finalize])


dnl The SUPERCLASS_USE_ASSOCIATIONS macro outputs the needed "use
dnl association" statements based on the definition of SUBCLASSES,
dnl which should be a space-delimited list of the subclasses.

define([SUPERCLASS_USE_ASSOCIATIONS],[
fortext([subclass], SUBCLASSES, [
  use subclass[]_Class
]) ])
```

```
dnl The SUPERCLASS_TYPE macro outputs a standard superclass type
dnl definition. It requires the following definitions:
dnl
dnl   - SUPERCLASS should already be defined to be the name of the
dnl     superclass.
dnl
dnl   - SUBCLASSES should already be defined to be a space-delimited
dnl     list of the subclasses.

define([SUPERCLASS_TYPE],[
  type SUPERCLASS[]_type
    type(character,80) :: Subclass
fortext([subclass], SUBCLASSES, [
    type(subclass[]_type) :: subclass
])
  end type SUPERCLASS[]_type
])


dnl Define the SUPERCLASS_DECLARATIONS macro, which is used internally
dnl by the SUPERCLASS_ROUTINE and SUPERCLASS_FUNCTION macros. This macro
dnl takes each group of three arguments, expands them in a declaration
dnl form like so:
dnl
dnl   $1 :: $2 ! $3
dnl
dnl and shifts them off the stack. It then continues with the next
dnl group of three arguments until there are no more arguments.

define([SUPERCLASS_DECLARATIONS],[
    ifelse($#, 0, ,
           $#, 1, [$1],
           $#, 2, [$1 :: $2],
           $#, 3, [$1 :: $2 [!] $3],
           [$1 :: $2 [!] $3 SUPERCLASS_DECLARATIONS(m4_shift(m4_shift(m4_shift($@))))]
    )])


dnl Define the SUPERCLASS_ARGUMENTS macro, which is used internally
dnl by the SUPERCLASS_ROUTINE and SUPERCLASS_FUNCTION macros. This macro
dnl takes each group of three arguments (in the same form as the
dnl SUPERCLASS_DECLARATIONS argument list) and pulls out the second
dnl argument (the actual variable) only, like so:
dnl
dnl   , $2
dnl
dnl and shifts the original three arguments off the stack. It then
dnl continues with the next group of three arguments until there are
dnl no more arguments. At the end of this operation, a variable list
dnl has been extracted in this form:
dnl
dnl   , var1, var2, var3, var4
```

```
define([SUPERCLASS_ARGUMENTS],
   [ifelse($#, 0, ,
           $#, 1, ,
           $#, 2, [, $2],
           $#, 3, [, $2],
           [, $2[]SUPERCLASS_ARGUMENTS(m4_shift(m4_shift(m4_shift($@))))])])


dnl Define the SUPERCLASS_ROUTINE macro, which expands into a complete
dnl subroutine for the superclass. This subroutine dynamically
dnl dispatches calls to the superclass to the correct subclass routine.
dnl There are some restrictions that must be true for this macro to
dnl behave correctly:
dnl
dnl   - SUPERCLASS should already be defined to be the name of the
dnl     superclass.
dnl
dnl   - SUBCLASSES should already be defined to be a space-delimited
dnl     list of the subclasses.
dnl
dnl   - The argument list for the superclass subroutine call must be the
dnl     same as the argument list for all of the subclass subroutine calls,
dnl     with the exception that the subclass calls are passed a component
dnl     of the superclass derived type corresponding to that subclass
dnl     instead of the entire superclass derived type.
dnl
dnl   - The superclass type must correspond to the type generated by the
dnl     the SUPERCLASS_TYPE macro.
dnl
dnl The arguments for the SUPERCLASS_ROUTINE macro are:
dnl
dnl   $1 - Generic Routine (and Interface) Name.
dnl   $(2 + n*3) - Type declaration for an additional variable to be added
dnl                to the argument list.
dnl   $(3 + n*3) - Variable name for an additional variable to be added
dnl                to the argument list.
dnl   $(4 + n*3) - Comment for an additional variable to be added to the
dnl                argument list.
dnl
dnl where n may be 0, 1, 2, etc., and the only required macro argument is
dnl the first one.

define([SUPERCLASS_ROUTINE],[
  pushdef([ROUT_NAME], [$1_[]SUPERCLASS])
  pushdef([VARLIST], [m4_shift($@)])
  pushdef([ARGS], [SUPERCLASS_ARGUMENTS(VARLIST)])

  subroutine ROUT_NAME (SUPERCLASS[]ARGS)

    type(SUPERCLASS[]_type) SUPERCLASS
    SUPERCLASS_DECLARATIONS(VARLIST)

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    select case (SUPERCLASS%Subclass)

  fortext([subclass], SUBCLASSES, [
    case ("subclass")

      call $1 (SUPERCLASS%subclass[]ARGS)
  ])
    case default

      write (6,*) 'Error: no ', SUPERCLASS%Subclass, ' in SUPERCLASS[]_Class.'

    end select

  end subroutine ROUT_NAME

  popdef([ROUT_NAME])
  popdef([VARLIST])
  popdef([ARGS])
])


dnl Define the SUPERCLASS_FUNCTION macro, which expands into a complete
dnl function for the superclass. This subroutine dynamically
dnl dispatches calls to the superclass to the correct subclass function.
dnl There are some restrictions that must be true for this macro to
dnl behave correctly:
dnl
dnl   - SUPERCLASS should already be defined to be the name of the
dnl     superclass.
dnl
dnl   - SUBCLASSES should already be defined to be a space-delimited
dnl     list of the subclasses.
dnl
dnl   - The argument list for the superclass function call must be the
dnl     same as the argument list for all of the subclass function calls,
dnl     with the exception that the subclass calls are passed a component
dnl     of the superclass derived type corresponding to that subclass
dnl     instead of the entire superclass derived type.
dnl
dnl   - The superclass type must correspond to the type generated by the
dnl     the SUPERCLASS_TYPE macro.
dnl
dnl The arguments for the SUPERCLASS_FUNCTION macro are:
dnl
dnl   $1 - Generic Function (and Interface) Name.
dnl   $2 - Type declaration for the function.
dnl   $(3 + n*3) - Type declaration for an additional variable to be added
dnl                to the argument list.
dnl   $(4 + n*3) - Variable name for an additional variable to be added
dnl                to the argument list.
dnl   $(5 + n*3) - Comment for an additional variable to be added to the
dnl                argument list.
dnl
```

```
dnl where n may be 0, 1, 2, etc., and the only required macro arguments are
dnl the first two.

define([SUPERCLASS_FUNCTION],[
  pushdef([FNCT_NAME], [$1_[]SUPERCLASS])
  pushdef([VARLIST], [m4_shift(m4_shift($@))])
  pushdef([ARGS], [SUPERCLASS_ARGUMENTS(VARLIST)])

  function FNCT_NAME (SUPERCLASS[]ARGS)

    type(SUPERCLASS[]_type) SUPERCLASS
    $2 :: $1, FNCT_NAME
    SUPERCLASS_DECLARATIONS(VARLIST)

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    select case (SUPERCLASS%Subclass)

  fortext([subclass], SUBCLASSES, [
    case ("subclass")

      FNCT_NAME = $1 (SUPERCLASS%subclass[]ARGS)
  ])
    case default

      write (6,*) 'Error: no ', SUPERCLASS%Subclass, ' in SUPERCLASS[]_Class.'

    end select

  end function FNCT_NAME

  popdef([FNCT_NAME])
  popdef([VARLIST])
  popdef([ARGS])
])


dnl Input text used to generate documentation:

dnl define([SUPERCLASS],[Matrix])
dnl define([SUBCLASSES],[One Two Three])
dnl
dnl module SUPERCLASS[]_Class
dnl
dnl   SUPERCLASS_USE_ASSOCIATIONS
dnl   SUPERCLASS_TYPE
dnl
dnl contains
dnl
dnl   SUPERCLASS_ROUTINE([Initialize],
dnl  [type(real)], [a], [The a variable],
dnl  [type(integer), intent(in)], [b], [The b variable])
dnl
dnl   SUPERCLASS_FUNCTION([Verify_State], [type(logical)],
```

```
dnl   [type(real)], [b], [The b variable])
dnl
dnl   SUPERCLASS_ROUTINE([Finalize],
dnl  [type(real)], [c], [The c variable],
dnl  [type(real)], [d], [The d variable])
dnl
dnl end module SUPERCLASS[]_Class
```

A.6 Unit Test m4 Macros

The main documentation of the Unit Test m4 Macros in section 6.6 contains additional explanation of this code listing.

```
dnl
dnl Author: Michael L. Hall
dnl         P.O. Box 1663, MS-D413, LANL
dnl         Los Alamos, NM 87545
dnl         ph: 505-665-4312
dnl         email: Hall@LANL.gov
dnl
dnl Created on: 12/07/98
dnl CVS Info:   $Id: unit_test.m4,v 1.12 2004/02/26 18:59:43 hall Exp $


dnl Define a macro that toggles output.

ifdef([UNIT_TEST],
      [define([TESTWRITE], [write])],
      [define([TESTWRITE], [! write])])


dnl Define a macro that comments out code if
dnl unit testing is not being done.

ifdef([UNIT_TEST],
      [define([IF_UNIT_TEST], [])],
      [define([IF_UNIT_TEST], [! ])])


dnl Define a macro that comments out code if
dnl unit testing is being done.

ifdef([UNIT_TEST],
      [define([IF_NOT_UNIT_TEST], [! ])],
      [define([IF_NOT_UNIT_TEST], [])])
```

A.7 Flags Module Code Listing

The main documentation of the Flags Module in § 6.7 on page 31 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 1/15/99
! CVS Info:   $Id: flags.F90,v 1.4 2008/09/26 00:04:27 hall Exp $

module Caesar_Flags_Module

  ! Start up with everything untyped and public.
  ! Note: this module contains no private information.

  implicit none
  public

  ! Initialization and finalization flags, used to set initial and
  ! final values for intrinsics.

  type(integer),     parameter :: initialize_integer_flag=0, &
                                    finalize_integer_flag=6622130
  type(real),        parameter :: initialize_real_flag=0.d0, &
                                    finalize_real_flag=662.2130d0
  type(logical),     parameter :: initialize_logical_flag=.false., &
                                    finalize_logical_flag=.false.
  type(character,9), parameter :: initialize_character_flag='Undefined', &
                                    finalize_character_flag='Finalized'

  ! Initialization flags, used to indicate when a derived type has
  ! been initialized.

  type(integer),     parameter :: uninitialized_flag=0, &
                                    initialized_flag=6622130

  ! Boundary condition face flags.

  type(integer),     parameter :: Internal_or_Periodic_BC=0, &
                                  Dirichlet_BC=1, Homogeneous_BC=-1, &
                                  Neumann_BC=2, Reflective_BC=-2, &
                                  Source_BC=3, Vacuum_BC=-3

  ! AMR flags to indicate large and small cells at a level-jump interface.
  ! These should not conflict with the boundary condition face flags.

  type(integer),     parameter :: AMR_Large_Cell_BC=4, &
                                  AMR_Small_Cell_BC=-4
```

```
end module Caesar_Flags_Module
```

A.7.1 Flags Class Unit Test Program

This lightly commented program performs a unit test on the Flags Class, which is described in § ?? on page ??.

```
program Unit_Test
  write(6,*) 'There is no Unit Test for the Flags Module.'
end
```

A.8 Numbers Module Code Listing

The main documentation of the Numbers Module in § 6.8 on page 32 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 6/1/86
! CVS Info:   $Id: numbers.F90,v 1.4 2009/09/04 01:58:44 hall Exp $

module Caesar_Numbers_Module

  ! Start up with everything untyped and public.
  ! Note: this module contains no private information.

  implicit none
  public

  ! Define the variables used in the place of numbers.

  ! Numbers 0-9.

  type(real), parameter :: zero=0.d0,  one=1.d0,  two=2.d0, three=3.d0, &
                           four=4.d0,  five=5.d0, six=6.d0, seven=7.d0, &
                           eight=8.d0, nine=9.d0

  ! Numbers 10-19.

  type(real), parameter :: ten=10.d0,      eleven=11.d0,    twelve=12.d0,   &
                           thirteen=13.d0, fourteen=14.d0,  fifteen=15.d0,  &
                           sixteen=16.d0,  seventeen=17.d0, eighteen=18.d0, &
                           nineteen=19.d0
```

```
  ! Numbers 20-100, by tens.

  type(real), parameter :: twenty=20.d0, thirty=30.d0, forty=40.d0,   &
                           fifty=50.d0,  sixty=60.d0,  seventy=70.d0, &
                           eighty=80.d0, ninety=90.d0, hundred=100.d0

  ! Fractions.

  type(real), parameter :: half=one/two,     third=one/three,   &
                           fourth=one/four,  fifth=one/five,    &
                           sixth=one/six,    seventh=one/seven, &
                           eighth=one/eight, ninth=one/nine,    &
                           tenth=one/ten

  ! Forms of pi.

  type(real), parameter ::                                            &
    pi=3.14159265358979323846264338327 9d0, sqrtpi=1.7724538509055d0,   &
    invpi=one/pi,    pisqr=pi*pi,          fourthirdspi=four*pi/three, &
    twopi=two*pi,    threepi=three*pi,     fourpi=four*pi,             &
    halfpi=pi/two,   thirdpi=pi/three,     fourthpi=pi/four,           &
    sqrtfourpi=two*sqrtpi

  ! Decimal multipliers.

  type(real), parameter :: deca=1.d1,    hecto=1.d2,  kilo=1.d3,    &
                           mega=1.d6,    giga=1.d9,   tera=1.d12,   &
                           peta=1.d15,   exa=1.d18,   zetta=1.d21,  &
                           yotta=1.d24,                             &
                           deci=1.d-1,   centi=1.d-2, milli=1.d-3,  &
                           micro=1.d-6,  nano=1.d-9,  pico=1.d-12,  &
                           femto=1.d-15, atto=1.d-18, zepto=1.d-21, &
                           yocto=1.d-24

end module Caesar_Numbers_Module
```

A.8.1 Numbers Class Unit Test Program

This lightly commented program performs a unit test on the Numbers Class, which is described in § **??** on page **??**.

```
program Unit_Test
  write(6,*) 'There is no Unit Test for the Numbers Module.'
end
```

Appendix B

Intrinsics Module Code Listing

The main documentation of the Intrinsics Module in chapter 7 on page 35 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 10/04/99
! CVS Info:   $Id: intrinsics.F90,v 1.3 2004/01/12 23:04:18 hall Exp $

module Caesar_Intrinsics_Module

  ! Global use associations.

  use Caesar_Real_Class
  use Caesar_Logical_Class
  use Caesar_Status_Class
  use Caesar_Integer_Class
  use Caesar_Character_Class

  ! Start up with everything untyped and public.
  ! Note: this module contains no private information.

  implicit none
  public

end module Caesar_Intrinsics_Module
```

B.1 Status Class Code Listing

The main documentation of the Status Class in § 7.1 on page 35 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 03/23/99
! CVS Info:   $Id: status.F90,v 7.7 2007/01/23 19:30:53 hall Exp $

module Caesar_Status_Class

  ! Global use associations - none.

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State
  public :: Assignment (=), Operator (==), Operator (/=), Consolidate, &
            Equal, Error, Get, Normal, Not_Equal, Set, Warning

  interface Initialize
    module procedure Initialize_Status
    module procedure Initialize_Status_Vector
  end interface

  interface Finalize
    module procedure Finalize_Status
    module procedure Finalize_Status_Vector
  end interface

  interface Valid_State
    module procedure Valid_State_Status
    module procedure Valid_State_Status_Vector
  end interface

  interface Assignment (=)
    module procedure Set_Status
    module procedure Get_Status_Output
    module procedure Consolidate_Status
  end interface

  interface Operator (==)
    module procedure Status_Equal_Status
    module procedure Status_Equal_Character
    module procedure Character_Equal_Status
  end interface

  interface Operator (/=)
    module procedure Status_Not_Equal_Status
```

```
    module procedure Status_Not_Equal_Character
    module procedure Character_Not_Equal_Status
  end interface

  interface Consolidate
    module procedure Consolidate_Status
  end interface

  interface Equal
    module procedure Status_Equal_Status
    module procedure Status_Equal_Character
    module procedure Character_Equal_Status
  end interface

  interface Error
    module procedure Error_Status
  end interface

  interface Get
    module procedure Get_Status_Output
  end interface

  interface Normal
    module procedure Normal_Status
  end interface

  interface Not_Equal
    module procedure Status_Not_Equal_Status
    module procedure Status_Not_Equal_Character
    module procedure Character_Not_Equal_Status
  end interface

  interface Set
    module procedure Set_Status
  end interface

  interface Warning
    module procedure Warning_Status
  end interface

  ! Public type definitions.

  public :: Status_type

  type Status_type
    private
    type(integer) :: status
  end type Status_type

  ! Global class variables.

  type(integer), parameter :: &
    NFlags=10,              &  ! Number of Flag types.
    output_length = 35,     &  ! Length of the output string.
```

```
    selector_length = 16,     &  ! Length of the selector string.
    severity_length = 7          ! Length of the severity string.

  type Flag       ! Status flag derived type.
    type(character,selector_length) :: selector
    type(character,output_length) :: output_string
    type(character,severity_length) :: severity
  end type Flag

  !-------------------------------------------------------------------------
  ! This is the main list of possible flags (or values for a status variable).
  !
  ! Note: when adding a new flag below, remember to:
  !
  !   - update NFlags above, and
  !   - add the new selector to the list in the
  !     Set_Status routine comments.
  !
  ! Flag order is unimportant.

  type(Flag), dimension(1:NFlags), parameter ::                                  &
    status_flag = (/                                                             &
          ! Selector            Output String                          Severity
      Flag('Unset           ', 'Status Initialized But Unset       ', 'Normal '), &
      Flag('Success         ', 'Successful Procedure Execution     ', 'Normal '), &
      Flag('Multiple Error  ', 'Multiple Error Types in Procedure  ', 'Error  '), &
      Flag('Multiple Warning', 'Multiple Warning Types in Procedure', 'Warning'), &
      Flag('Memory Warning  ', 'Memory Allocation Warning          ', 'Warning'), &
      Flag('Memory Error    ', 'Memory Allocation Error            ', 'Error  '), &
      Flag('File Error      ', 'File Access Error                  ', 'Error  '), &
      Flag('CGNS Error      ', 'CGNS Error                         ', 'Error  '), &
      Flag('CGNS No Node    ', 'CGNS NODE_NOT_FOUND Error          ', 'Error  '), &
      Flag('CGNS Bad Path   ', 'CGNS INCORRECT_PATH Error          ', 'Error  ')  &
    /)
  !-------------------------------------------------------------------------

contains
```

The Status Class contains the following routines which are listed in separate sections:

Normal_Status (§ B.1.12, page 226)

Set_Status (§ B.1.13, page 227)

Get_Status_Output (§ B.1.11, page 226)

Status_Equal_Status (§ B.1.15, page 228)

Status_Not_Equal_Status (§ B.1.17, page 230)

Status_Equal_Character (§ B.1.14, page 228)

Status_Not_Equal_Character (§ B.1.16, page 229)

Character_Equal_Status (§ B.1.7, page 221)

Character_Not_Equal_Status (§ B.1.8, page 221)

Consolidate_Status (§ B.1.9, page 222)

end module Caesar_Status_Class

B.1.1 Initialize_Status Procedure

The main documentation of the Initialize_Status Procedure in § 7.1.1 on page 36 contains additional explanation of this code listing.

```
  subroutine Initialize_Status (S)

    ! Output variable.

    type(Status_type), intent(out) :: S  ! Status to be initialized.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Initializations.

    S = 'Unset'

    ! Verify guarantees.

    VERIFY(Valid_State(S),1)  ! S is now valid.

    return
  end subroutine Initialize_Status
```

B.1.2 Initialize_Status_Vector Procedure

The main documentation of the Initialize_Status_Vector Procedure in § 7.1.2 on page 36 contains additional explanation of this code listing.

```
subroutine Initialize_Status_Vector (S)

  ! Output variable.

  ! Status vector to be initialized:
  type(Status_type), dimension(:), intent(out) :: S

  ! Internal variable.

  type(integer) :: i  ! Loop counter.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Initializations.

  do i = 1, SIZE(S)
    call Initialize (S(i))
  end do

  ! Verify guarantees.

  VERIFY(Valid_State(S),1)  ! S is now valid.

  return
end subroutine Initialize_Status_Vector
```

B.1.3 Finalize_Status Procedure

The main documentation of the Finalize_Status Procedure in § 7.1.3 on page 37 contains additional explanation of this code listing.

```
subroutine Finalize_Status (S)

  ! Input/Output variable.

  type(Status_type), intent(inout) :: S  ! Status to be finalized.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(S),1)         ! S is valid.

  ! Finalizations.

  S%status = 0

  ! Verify guarantees.
```

```
    VERIFY(.not.(Valid_State(S)),1)  ! S is no longer valid.

    return
  end subroutine Finalize_Status
```

B.1.4 Finalize_Status_Vector Procedure

The main documentation of the Finalize_Status_Vector Procedure in § 7.1.4 on page 37 contains additional explanation of this code listing.

```
  subroutine Finalize_Status_Vector (S)

    ! Input/Output variable.

    ! Status vector to be finalized:
    type(Status_type), dimension(:), intent(inout) :: S

    ! Internal variable.

    type(integer) :: i  ! Loop counter.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S),1)         ! S is valid.

    ! Finalizations.

    do i = 1, SIZE(S)
      call Finalize (S(i))
    end do

    ! Verify guarantees.

    VERIFY(.not.(Valid_State(S)),1)  ! S is no longer valid.

    return
  end subroutine Finalize_Status_Vector
```

B.1.5 Valid_State_Status Procedure

The main documentation of the Valid_State_Status Procedure in § 7.1.5 on page 37 contains additional explanation of this code listing.

```
  function Valid_State_Status (S) result(Valid)

    ! Input variable.
```

```
    type(Status_type), intent(in) :: S  ! Status to be checked.

    ! Output variable.

    type(logical) :: Valid              ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Make sure the status variable is in range.

    Valid = Valid .and. S%status <= NFlags
    Valid = Valid .and. S%status >= 1

    return
  end function Valid_State_Status
```

B.1.6 Valid_State_Status_Vector Procedure

The main documentation of the Valid_State_Status_Vector Procedure in § 7.1.6 on page 37 contains additional explanation of this code listing.

```
  function Valid_State_Status_Vector (S) result(Valid)

    ! Input variable.

    type(Status_type), dimension(:), intent(in) :: S  ! Status to be checked.

    ! Output variable.

    type(logical) :: Valid   ! Logical state.

    ! Internal variable.

    type(integer) :: i       ! Loop counter.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check each element.

    do i = 1, SIZE(S)
      Valid = Valid .and. Valid_State(S(i))
    end do

    return
```

```
  end function Valid_State_Status_Vector
```

B.1.7 Character_Equal_Status Procedure

The main documentation of the Character_Equal_Status Procedure in § 7.1.7 on page 38 contains additional explanation of this code listing.

```
  function Character_Equal_Status (C, SS) result(Equal)

    ! Input variables.

    type(Status_type), intent(in) :: SS  ! Status variable to be compared.
    type(character,*), intent(in) :: C   ! Selector flag string to be compared.

    ! Output variable.

    type(logical) :: Equal               ! Equality boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(SS),1)   ! SS is valid.

    ! Check equality.

    Equal = status_flag(SS%status)%selector == C

    ! Verify guarantees.

    ! Equal should be what it was set to.
    VERIFY(Equal .eqv. (status_flag(SS%status)%selector == C),2)

    return
  end function Character_Equal_Status
```

B.1.8 Character_Not_Equal_Status Procedure

The main documentation of the Character_Not_Equal_Status Procedure in § 7.1.8 on page 38 contains additional explanation of this code listing.

```
  function Character_Not_Equal_Status (C, SS) result(Not_Equal)

    ! Input variable.

    type(Status_type), intent(in) :: SS  ! Status variable to be compared.
    type(character,*), intent(in) :: C   ! Selector flag string to be compared.

    ! Output variable.
```

```
    type(logical) :: Not_Equal               ! Nonequality boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(SS),1)   ! SS is valid.

    ! Check nonequality.

    Not_Equal = status_flag(SS%status)%selector /= C

    ! Verify guarantees.

    ! Not_Equal should be what it was set to.
    VERIFY(Not_Equal .eqv. (status_flag(SS%status)%selector /= C),2)

    return
  end function Character_Not_Equal_Status
```

B.1.9 Consolidate_Status Procedure

The main documentation of the Consolidate_Status Procedure in § 7.1.9 on page 39 contains additional explanation of this code listing.

```
  subroutine Consolidate_Status (Consolidated_S, Multiple_S)

    ! Input variable.

    ! Vector of status variables to be consolidated:
    type(Status_type), intent(in), dimension(:) :: Multiple_S

    ! Output variable.

    type(Status_type), intent(out) :: Consolidated_S  ! Consolidated status.

    ! Internal variable.

    type(integer) :: i  ! Loop counter.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Multiple_S),1)   ! Multiple_S is valid.

    ! The following table shows the value of Consolidated_S after it has been
    ! combined with a single value from the vector Multiple_S, based on the
    ! previous value of Consolidated_S:
    !
    !                                          Multiple_S(i)
```

```
!
!                             Unset    Success  ME  MW  Error  Warning
!                           +-----------------------------------------+
!                   Unset   | Unset    Success  ME  MW  Error  Warning |
!                   Success | Success  Success  ME  MW  Error  Warning |
!                   ME      | ME       ME       ME  ME  ME     ME      |
!   Consolidated_S  MW      | MW       MW       ME  MW  ME     MW      |
!     (previous)    Error   | Error    Error    ME  ME  ME*    ME      |
!                   Warning | Warning  Warning  ME  MW  ME     MW*     |
!                           +-----------------------------------------+
!
!   ME: Multiple Error
!   MW: Multiple Warning
!   *: Multiple Error or Warning is only set
!      if the two errors or warnings differ.
!
! Notice that this matrix is symmetric.

! Start out Unset.

Consolidated_S = 'Unset'

! Loop over Multiple_S vector.

do i = 1, SIZE(Multiple_S)

  ! Switch on Multiple_S(i).

  select case (status_flag(Multiple_S(i)%status)%selector)

  ! Multiple_S(i) = 'Unset'
  !
  !   Consolidated_S (old):  Unset    Success  ME  MW  Error  Warning
  !   Consolidated_S (new):  Unset    Success  ME  MW  Error  Warning

  case ('Unset')

    ! Do not modify Consolidated_S.

  ! Multiple_S(i) = 'Success'
  !
  !   Consolidated_S (old):  Unset    Success  ME  MW  Error  Warning
  !   Consolidated_S (new):  Success  Success  ME  MW  Error  Warning

  case ('Success')

    if (status_flag(Consolidated_S%status)%selector == 'Unset') then
      Consolidated_S = 'Success'
    end if

  ! Multiple_S(i) = 'Multiple Error'
  !
  !   Consolidated_S (old):  Unset    Success  ME  MW  Error  Warning
  !   Consolidated_S (new):  ME       ME       ME  ME  ME     ME
```

```
case ('Multiple Error')

  Consolidated_S = 'Multiple Error'

! Multiple_S(i) = 'Multiple Warning'
!
!   Consolidated_S (old):  Unset    Success  ME  MW  Error  Warning
!   Consolidated_S (new):  MW       MW       ME  MW  ME     MW

case ('Multiple Warning')

  if (Error(Consolidated_S)) then
    Consolidated_S = 'Multiple Error'
  else
    Consolidated_S = 'Multiple Warning'
  end if

case default

  ! Multiple_S(i) = 'Error'
  !
  !   Consolidated_S (old):  Unset    Success  ME  MW  Error  Warning
  !   Consolidated_S (new):  Error    Error    ME  ME  ME*    ME

  if (Error(Multiple_S(i))) then

    if (Error(Consolidated_S)) then
      if (Consolidated_S /= Multiple_S(i)) then
        Consolidated_S = 'Multiple Error'
      end if
    else if (Warning(Consolidated_S)) then
      Consolidated_S = 'Multiple Error'
    else
      Consolidated_S = Multiple_S(i)
    end if

  ! Multiple_S(i) = 'Warning'
  !
  !   Consolidated_S (old):  Unset    Success  ME  MW  Error  Warning
  !   Consolidated_S (new):  Warning  Warning  ME  MW  ME     MW*

  else if (Warning(Multiple_S(i))) then

    if (Error(Consolidated_S)) then
      Consolidated_S = 'Multiple Error'
    else if (Warning(Consolidated_S)) then
      if (Consolidated_S /= Multiple_S(i)) then
        Consolidated_S = 'Multiple Warning'
      end if
    else
      Consolidated_S = Multiple_S(i)
    end if
```

```
        ! This condition should not be hit.

        else

          write (6,*) 'Consolidate_Status: Impossible Status Combination Hit.'

        end if

      end select

    ! End of loop over Multiple_S vector.

    end do

    ! Verify guarantees.

    VERIFY(Valid_State(Consolidated_S),1)   ! Consolidated_S is valid.

    !VERIFY(,2)

    return
  end subroutine Consolidate_Status
```

B.1.10 Error_Status Procedure

The main documentation of the Error_Status Procedure in § 7.1.10 on page 39 contains additional explanation of this code listing.

```
  function Error_Status (S) result(Error)

    ! Input variable.

    type(Status_type), intent(in) :: S  ! Status to be checked.

    ! Output variable.

    type(logical) :: Error              ! Error condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S),1)   ! S is valid.

    ! Set error boolean.

    Error = status_flag(S%status)%severity == 'Error'

    ! Verify guarantees.

    ! Error should be what it was set to.
    VERIFY(Error .eqv. (status_flag(S%status)%severity == 'Error'),2)
```

```
    return
  end function Error_Status
```

B.1.11 Get_Status_Output Procedure

The main documentation of the Get_Status_Output Procedure in § 7.1.11 on page 40 contains additional explanation of this code listing.

```
  subroutine Get_Status_Output (Status_String, S)

    ! Input variable.

    type(Status_type), intent(in) :: S  ! Status.

    ! Output variable.

    ! Output string for this Status.
    type(character,*), intent(out) :: Status_String

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S),1)   ! S is valid.

    ! Determine which output string is to be set.

    Status_String = status_flag(S%status)%output_string

    ! Verify guarantees.

    ! Status_String should be what it was set to.
    VERIFY(Status_String == status_flag(S%status)%output_string,2)

    return
  end subroutine Get_Status_Output
```

B.1.12 Normal_Status Procedure

The main documentation of the Normal_Status Procedure in § 7.1.12 on page 40 contains additional explanation of this code listing.

```
  function Normal_Status (S) result(Normal)

    ! Input variable.

    type(Status_type), intent(in) :: S  ! Status to be checked.
```

```
    ! Output variable.

    type(logical) :: Normal             ! Normal condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S),1)   ! S is valid.

    ! Set normal boolean.

    Normal = status_flag(S%status)%severity == 'Normal'

    ! Verify guarantees.

    ! Normal should be what it was set to.
    VERIFY(Normal .eqv. (status_flag(S%status)%severity == 'Normal'),2)

    return
  end function Normal_Status
```

B.1.13 Set_Status Procedure

The main documentation of the Set_Status Procedure in § 7.1.13 on page 40 contains additional explanation of this code listing.

```
  subroutine Set_Status (S, Selector_Flag)

    ! Input variable.

    ! String to select status value.
    type(character,*), intent(in) :: Selector_Flag

    ! Output variable.

    type(Status_type), intent(out) :: S  ! Status to be set.

    ! Internal variable.

    type(integer) :: i  ! Loop counter.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    ! Selector_Flag must be one of the possible flags.
    VERIFY(ANY(Selector_Flag == status_flag%selector),1)

    ! Determine which flag is to be set.

    do i = 1, NFlags
```

```
      if (Selector_Flag == status_flag(i)%selector) S%status = i
    end do

    ! Verify guarantees.

    VERIFY(Valid_State(S),1)   ! S is now valid.

    return
  end subroutine Set_Status
```

B.1.14 Status_Equal_Character Procedure

The main documentation of the Status_Equal_Character Procedure in § 7.1.14 on page 41 contains additional explanation of this code listing.

```
  function Status_Equal_Character (S, C) result(Equal)

    ! Input variables.

    type(Status_type), intent(in) :: S  ! Status variable to be compared.
    type(character,*), intent(in) :: C   ! Selector flag string to be compared.

    ! Output variable.

    type(logical) :: Equal               ! Equality boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S),1)   ! S is valid.

    ! Check equality.

    Equal = status_flag(S%status)%selector == C

    ! Verify guarantees.

    ! Equal should be what it was set to.
    VERIFY(Equal .eqv. (status_flag(S%status)%selector == C),2)

    return
  end function Status_Equal_Character
```

B.1.15 Status_Equal_Status Procedure

The main documentation of the Status_Equal_Status Procedure in § 7.1.15 on page 41 contains additional explanation of this code listing.

```
  function Status_Equal_Status (S1, S2) result(Equal)

    ! Input variables.

    type(Status_type), intent(in) :: S1, S2  ! Status variables to be compared.

    ! Output variable.

    type(logical) :: Equal                   ! Equality boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S1),1)   ! S1 is valid.
    VERIFY(Valid_State(S2),1)   ! S2 is valid.

    ! Check equality.

    Equal = S1%status == S2%status

    ! Verify guarantees.

    ! Equal should be what it was set to.
    VERIFY(Equal .eqv. (S1%status == S2%status),2)

    return
  end function Status_Equal_Status
```

B.1.16 Status_Not_Equal_Character Procedure

The main documentation of the Status_Not_Equal_Character Procedure in § 7.1.16 on page 42 contains additional explanation of this code listing.

```
  function Status_Not_Equal_Character (S, C) result(Not_Equal)

    ! Input variables.

    type(Status_type), intent(in) :: S  ! Status variable to be compared.
    type(character,*), intent(in) :: C  ! Selector flag string to be compared.

    ! Output variable.

    type(logical) :: Not_Equal          ! Nonequality boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S),1)   ! S is valid.

    ! Check nonequality.
```

```
    Not_Equal = status_flag(S%status)%selector /= C

    ! Verify guarantees.

    ! Not_Equal should be what it was set to.
    VERIFY(Not_Equal .eqv. (status_flag(S%status)%selector /= C),2)

    return
  end function Status_Not_Equal_Character
```

B.1.17 Status_Not_Equal_Status Procedure

The main documentation of the Status_Not_Equal_Status Procedure in § 7.1.17 on page 42 contains additional explanation of this code listing.

```
  function Status_Not_Equal_Status (S1, S2) result(Not_Equal)

    ! Input variables.

    type(Status_type), intent(in) :: S1, S2  ! Status variables to be compared.

    ! Output variable.

    type(logical) :: Not_Equal               ! Nonequality boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S1),1)   ! S1 is valid.
    VERIFY(Valid_State(S2),1)   ! S2 is valid.

    ! Check nonequality.

    Not_Equal = S1%status /= S2%status

    ! Verify guarantees.

    ! Not_Equal should be what it was set to.
    VERIFY(Not_Equal .eqv. (S1%status /= S2%status),2)

    return
  end function Status_Not_Equal_Status
```

B.1.18 Warning_Status Procedure

The main documentation of the Warning_Status Procedure in § 7.1.18 on page 42 contains additional explanation of this code listing.

```
  function Warning_Status (S) result(Warning)

    ! Input variable.

    type(Status_type), intent(in) :: S  ! Status to be checked.

    ! Output variable.

    type(logical) :: Warning            ! Warning condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(S),1)   ! S is valid.

    ! Set warning boolean.

    Warning = status_flag(S%status)%severity == 'Warning'

    ! Verify guarantees.

    ! Warning should be what it was set to.
    VERIFY(Warning .eqv. (status_flag(S%status)%severity == 'Warning'),2)

    return
  end function Warning_Status
```

B.1.19 Status Class Unit Test Program

This lightly commented program performs a unit test on the Status Class, which is described in § 7.1 on page 35.

```
program Unit_Test

  use Caesar_Status_Class
  implicit none

  type(integer), parameter :: NStats=12
  type(Status_type), dimension(NStats) :: Status
  type(Status_type) :: Final_Status
  type(character,36) :: status_string
  type(integer) :: i, j

  ! Initialize status.

  call Initialize (Status)
  call Initialize (Final_Status)

  ! Check state of status.
```

```
VERIFY(Valid_State(Status),0)
VERIFY(Valid_State(Final_Status),0)

! Testing statements.

Status(2)  = 'Memory Error'
Status(3)  = 'Memory Error'
Status(4)  = 'Success'
Status(5)  = 'Memory Warning'
Status(6)  = 'Unset'
Status(7)  = 'Success'
Status(8)  = 'Memory Warning'
Status(9)  = 'Multiple Warning'
Status(10) = 'Success'
Status(11) = 'Multiple Error'
Status(12) = 'Multiple Warning'

write (6,101) 'Assignment tests:'

do i = 1, NStats
  if (Error(Status(i))) write (6,100,advance='no') 'Error: '
  if (Warning(Status(i))) write (6,100,advance='no') 'Warning: '
  if (Normal(Status(i))) write (6,100,advance='no') 'Normal: '
  status_string = Status(i)
  write (6,*) status_string
end do

write (6,101) 'Consolidation tests:'

do i = 1, NStats
  do j = i, NStats
    Final_Status = Status(i:j)
    if (Error(Final_Status)) write (6,100,advance='no') 'Error: '
    if (Warning(Final_Status)) write (6,100,advance='no') 'Warning: '
    if (Normal(Final_Status)) write (6,100,advance='no') 'Normal: '
    status_string = Final_Status
    write (6,*) status_string
  end do
  write (6,*)
end do

! Format statement.

100 format (a)
101 format (/,a,/)

! Check state of status.

VERIFY(Valid_State(Status),0)
VERIFY(Valid_State(Final_Status),0)

! Finalize status.

call Finalize (Status)
```

```
  call Finalize (Final_Status)

end
```

B.2 Real Class Code Listing

The main documentation of the Real Class in § 7.2 on page 43 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 12/04/98
! CVS Info:   $Id: real.F90,v 8.14 2008/09/29 21:48:37 hall Exp $

module Caesar_Real_Class

  ! Global use associations.

  use Caesar_Status_Class
  use Caesar_Logical_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Valid_State_NP
  public :: Operator(.VeryClose.)
  public :: MaxVal, MinVal, SUM, VeryClose

  REPLICATE_INTERFACE([Initialize], [Initialize_Real])

  REPLICATE_INTERFACE([Finalize], [Finalize_Real])

  REPLICATE_INTERFACE([Valid_State], [Valid_State_Real_P])
  REPLICATE_INTERFACE([Valid_State_NP], [Valid_State_Real_NP])

  interface MaxVal
    module procedure MaxVal_Real_Scalar
  end interface

  interface MinVal
    module procedure MinVal_Real_Scalar
  end interface
```

```
  interface SUM
    module procedure SUM_Real_Scalar
  end interface

  REPLICATE_INTERFACE([VeryClose], [VeryClose_Real])

  forloop([Dim],[0],[7],[
    pushdef([VeryClose_Real_Dim], expand(VeryClose_Real_Dim))
    interface OPERATOR (.VeryClose.)
      module procedure VeryClose_Real_Dim
    end interface
    popdef([VeryClose_Real_Dim])
  ])

contains
```

The Real_Class contains the following routines which are listed in separate sections:

end module Caesar_Real_Class

B.2.1 Initialize_Real Procedure

The main documentation of the Initialize_Real Procedure in § 7.2.1 on page 43 contains additional explanation of this code listing.

```
  subroutine Initialize_Real_0 (R, status)

    ! Use association information.

    use Caesar_Flags_Module, only: initialize_real_flag

    ! Output variables.

    type(real), intent(out) :: R        ! Variable to be initialized.
    type(Status_type), intent(out), optional :: status ! Exit status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.
```

```
  ! Initialize to flag value.

  R = initialize_real_flag

  ! No errors for initialization possible for scalars.

  if (PRESENT(status)) status = 'Success'

  ! Verify guarantees - none.

  return
end subroutine Initialize_Real_0

define([REPLICATE_ROUTINE],[
  subroutine Initialize_Real_$1 (R REP_ARGS([dim[]i]), status)

    ! Use association information.

    use Caesar_Flags_Module, only: initialize_real_flag

    ! Input variables.

    REP_DECLARE([type(integer), intent(in)], [dim[]i])  ! Array dimensions.

    ! Input/Output variable.

    type(real,$1) :: R                 ! Variable to be initialized.

    ! Output variable.

    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variable.

    type(integer) :: allocate_status   ! Allocation Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    ! The association status of a unallocated pointer is officially
    ! undefined according to the Fortran standard. With most compilers,
    ! the status is unassociated.
    ifelse(COMPILER, NAGWare,
      [], [
      VERIFY(.not.ASSOCIATED(R), 0)   ! R starts out unassociated.
    ])

    ! Allocation (for arrays only).

    REP_ALLOCATE([R], [dim[]i], [allocate_status])

    ! Initialize to flag value.
```

```
    R = initialize_real_flag

    ! Verify guarantees and/or set status flag.

    if (PRESENT(status)) then
      WARN_IF(allocate_status /= 0, 3)  ! Allocation error check.
      WARN_IF(.not.ASSOCIATED(R),3)     ! R is now associated.
      if (allocate_status == 0 .and. ASSOCIATED(R)) then
        status = 'Success'
      else
        status = 'Memory Error'
      end if
    else
      VERIFY(allocate_status == 0, 0)  ! Allocation error check.
      VERIFY(ASSOCIATED(R),0)          ! R is now associated.
    end if

    return
  end subroutine Initialize_Real_$1
])

REPLICATE_ARRAYS
```

B.2.2 Finalize_Real Procedure

The main documentation of the Finalize_Real Procedure in § 7.2.2 on page 44 contains additional explanation of this code listing.

```
subroutine Finalize_Real_0 (R, status)

  ! Use association information.

  use Caesar_Flags_Module, only: finalize_real_flag

  ! Input/Output variable.

  type(real), intent(inout) :: R    ! Variable to be finalized.

  ! Output variable.

  type(Status_type), intent(out), optional :: status  ! Exit status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Finalization.

  R = finalize_real_flag

  ! No errors for finalization possible for scalars.
```

```
    if (PRESENT(status)) status = 'Success'

    ! Verify guarantees - none.

    return
  end subroutine Finalize_Real_0

  define([REPLICATE_ROUTINE],[
    subroutine Finalize_Real_$1 (R, status)

      ! Input/Output variable.

      type(real,$1) :: R                 ! Variable to be finalized.

      ! Output variable.

      type(Status_type), intent(out), optional :: status  ! Exit status.

      ! Internal variable.

      type(integer) :: deallocate_status ! Deallocation Status.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(ASSOCIATED(R),0)           ! R should be associated.

      ! Deallocation.

      DEALLOCATE(R, stat=deallocate_status)

      ! Finalization and nullification.

      NULLIFY(R)

      ! Verify guarantees and/or set status flag.

      if (PRESENT(status)) then
        WARN_IF(deallocate_status /= 0, 3)  ! Deallocation error check.
        WARN_IF(ASSOCIATED(R),3)            ! R is now unassociated.
        if (deallocate_status == 0 .and. .not.ASSOCIATED(R)) then
          status = 'Success'
        else
          status = 'Memory Error'
        end if
      else
        VERIFY(deallocate_status == 0, 0)  ! Deallocation error check.
        VERIFY(.not.ASSOCIATED(R), 0)      ! R is now unassociated.
      end if

      return
    end subroutine Finalize_Real_$1
  ])
```

```
REPLICATE_ARRAYS
```

B.2.3 Valid_State_Real Procedure

The main documentation of the Valid_State_Real Procedure in § 7.2.3 on page 44 contains additional explanation of this code listing.

```
! Turn off checking which involves division by zero for some compilers
! that allow error trapping.

! For Suns, you could either
!   - not set DIVISION_BY_ZERO and use no compiler flags, or
!   - set DIVISION_BY_ZERO and use -ftrap=%none.
! For Intel/NAGWare, you could either
!   - not set DIVISION_BY_ZERO and use no compiler flags, or
!   - set DIVISION_BY_ZERO and use -ieee=full.
ifelse(
  ARCHITECTURE, Sun,
    [],
  ARCHITECTURE, SGI,
    [define([DIVISION_BY_ZERO],1)],
  ARCHITECTURE, Intel, [
    ifelse(
      COMPILER, NAGWare,
        [],
        [define([DIVISION_BY_ZERO],1)]
    )],
  ARCHITECTURE, Apple,
    [define([DIVISION_BY_ZERO],1)]
)

define([REPLICATE_ROUTINE],[
  ifelse(POINTER_TOGGLE, [TRUE], [
    pushdef([TYPE], [real,$1])
    pushdef([Valid_State_Real_P_DIM], expand(Valid_State_Real_P_$1))
    pushdef([POINTER_ONLY], [])
  ],[
    pushdef([TYPE], [real,$1,np])
    pushdef([Valid_State_Real_P_DIM], expand(Valid_State_Real_NP_$1))
    pushdef([POINTER_ONLY], [!])
  ])

  function Valid_State_Real_P_DIM (R) result(Valid)

    ! Use association information.

    SCALAR_ONLY use Caesar_Flags_Module, only: finalize_real_flag
    SCALAR_ONLY use Caesar_Logical_Class, only: ALL

    ! Input variable.
```

```
type(TYPE) :: R               ! Variable to be checked.

! Output variable.

type(logical) :: Valid        ! Logical state.

! Internal variables.

type(real) :: one, ten, zero  ! Numbers are not parameterized
                              ! to fool smart compilers.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Set numbers carefully (so that the compiler
! doesn't know that zero=0).

ten = 1.d1
one = 1.d0
zero = one - one

! Start out true.

Valid = .true.

! First, make sure that the variable has been allocated.

POINTER_ONLY ARRAY_ONLY Valid = Valid .and. ASSOCIATED(R)
POINTER_ONLY ARRAY_ONLY if (.not.Valid) return

! Make sure the variable has not been finalized.

SCALAR_ONLY Valid = Valid .and. R /= finalize_real_flag

! Check for Infs. This check determines whether R(1-e) = R, where e
! is a small number. This should only be true if R = 0 or if R is
! not a valid number.
!
! Pass Table:
!             Intel  Intel   Intel    Intel  Apple   Sun   SGI   IBM
!             Lahey  Absoft  NAGWare  PGI    Absoft
!   Infinity  Fail   Fail    Fail     Fail   Fail    Fail  Fail  Fail
!  -Infinity  Fail   Fail    Fail     Fail   Fail    Fail  Fail  Fail
!   NaN       Pass   Pass    Pass     Pass   Pass    Pass  Pass  Pass

Valid = Valid .and. ALL(R == zero .or. R*(one - ten*EPSILON(one)) /= R)
TESTWRITE (6,100) 'Test 1, R(1-e) = R     ==>', &
  IF_UNIT_TEST ALL(R == zero .or. R*(one - ten*EPSILON(one)) /= R)

! For IEEE-conforming reals, the following is (supposedly) a check
! for NaNs.
!
! Pass Table (test 2):
!             Intel  Intel   Intel    Intel  Apple   Sun   SGI   IBM
!             Lahey  Absoft  NAGWare  PGI    Absoft
```

```
!   Infinity  Pass   Pass    Pass     Pass   Pass    Pass  Pass  Pass
!  -Infinity  Pass   Pass    Pass     Pass   Pass    Pass  Pass  Pass
!   NaN       Pass   Fail    Fail     Pass   Fail    Fail  Pass  Fail
!
! NaN behavior details:
!
! Intel/Lahey (pre-L6.20c), Intel/Absoft, Intel/NAGWare
! (with -ieee=full), Apple/Absoft and Sun (with -ftrap=%none):
!   Fail on tests 2 and 2a, but pass on 2b, for scalars.
!   Fail on test 2, but pass on 2a and 2b, for arrays.
! Intel/Lahey:
!   With the L6.20c compiler, the behavior is:
!   Fail on test2a, but pass on 2 and 2b, for scalars.
!   Passes on tests 2, 2a and 2b, for arrays.
!   In other words, since test 2 passes for both scalars and arrays,
!   this is not a good test for the Lahey compiler.
! Sun:
!   Changed behavior after the 107356-02 patch; the new behavior
!   is reflected here. So, with the Sun 5.0 compiler with 107377-02
!   and 107356-02 patches, and the -ftrap=%none flag set to disable
!   exception trapping, this now works as a check for NaNs.
! IBM:
!   Fails on tests 2 and 2a, but passes on 2b. (Recheck behavior with
!   arrays to see if it is the same as some other compilers above if
!   IBM access is regained.)

Valid = Valid .and. ALL(R == R)
TESTWRITE (6,100) 'Test 2, R == R         ==>', ALL(R == R)
TESTWRITE (6,100) 'Test 2a, .not.(R /= R) ==>', .not. ALL((R /= R))
TESTWRITE (6,100) 'Test 2b, .not.(R < R)  ==>', .not. ALL((R < R))

! Create an infinity and check to verify inequality.
!
! Pass Table:
!             Intel  Intel   Intel    Intel  Apple   Sun   SGI   IBM
!             Lahey  Absoft  NAGWare  PGI    Absoft
!   Infinity  Fail   Fail    Fail     Fail   Fail    Fail  Fail  Fail
!  -Infinity  Pass   Pass    Pass     Pass   Pass    Pass  Pass  Pass
!   NaN       Pass   Pass    Pass     Fail   Pass    Pass  Pass  Pass

ifdef([DIVISION_BY_ZERO],[
  Valid = Valid .and. ALL(one/zero /= R)
  TESTWRITE (6,100) 'Test 3, Infinity /= R  ==>', ALL(one/zero /= R)
])

! Create a negative infinity and check to verify inequality.
!
! Pass Table:
!            Intel  Intel   Intel    Intel  Apple   Sun   SGI   IBM
!            Lahey  Absoft  NAGWare  PGI    Absoft
!  Infinity  Pass   Pass    Pass     Pass   Pass    Pass  Pass  Pass
! -Infinity  Fail   Fail    Fail     Fail   Fail    Fail  Fail  Fail
!  NaN       Pass   Pass    Pass     Fail   Pass    Pass  Pass  Pass
```

```
    ifdef([DIVISION_BY_ZERO],[
      Valid = Valid .and. ALL(-one/zero /= R)
      TESTWRITE (6,100) 'Test 4, -Infinity /= R ==>', ALL(-one/zero /= R)
    ])

    ! Create a NaN and check to verify inequality.
    !
    ! Pass Table:
    !            Intel  Intel   Intel    Intel  Apple   Sun   SGI   IBM
    !            Lahey  Absoft  NAGWare  PGI    Absoft
    !  Infinity  Pass   Pass    Pass     Fail   Pass    Pass  Pass  Pass
    ! -Infinity  Pass   Pass    Pass     Fail   Pass    Pass  Pass  Pass
    !  NaN       Pass   Pass    Pass     Fail   Pass    Pass  Pass  Pass
    !
    ! This test does not work for any compiler yet. It seems to work for
    ! Intel/PGI from the above table, but the test also fails for all valid
    ! real numbers, so it can't be used.

    ifdef([DIVISION_BY_ZERO],[
      ifelse(
        COMPILER, PGI,
          [], [
        Valid = Valid .and. ALL(zero/zero /= R)
        TESTWRITE (6,100) 'Test 5, NaN /= R       ==>', ALL(zero/zero /= R)
      ])
    ])

    ! Check the top of the range.
    !
    ! Pass Table:
    !             Intel  Intel   Intel    Intel  Apple   Sun   SGI   IBM
    !             Lahey  Absoft  NAGWare  PGI    Absoft
    !   Infinity  Fail   Fail    Fail     Fail   Fail    Fail  Fail  Fail
    !  -Infinity  Pass   Pass    Pass     Pass   Pass    Pass  Pass  Pass
    !   NaN       Fail   Fail    Fail     Pass   Fail    Fail  Fail  Fail

    Valid = Valid .and. ALL(R <= HUGE(R))
    TESTWRITE (6,100) 'Test 6, R <= HUGE(R)   ==>', ALL(R <= HUGE(R))

    ! Note that there is no explicit check for the bottom of the
    ! range, since there is no F90 intrinsic that returns the lowest
    ! negative number that a real can take.

    ! Format statement.

    IF_UNIT_TEST 100 format (2x, a, 1x, l1)

    return
  end function Valid_State_Real_P_DIM

  popdef([TYPE])
  popdef([Valid_State_Real_P_DIM])
  popdef([POINTER_ONLY])
])
```

```
define([POINTER_TOGGLE], [TRUE])
REPLICATE

define([POINTER_TOGGLE], [FALSE])
REPLICATE
```

B.2.4 MaxVal_Real_Scalar Procedure

The main documentation of the MaxVal_Real_Scalar Procedure in § 7.2.4 on page 45 contains additional explanation of this code listing.

```
function MaxVal_Real_Scalar (R)

  ! Input variable.

  type(real), intent(in) :: R

  ! Output variable.

  type(real) :: MaxVal_Real_Scalar

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! MaxVal_Real_Scalar is equal to R.

  MaxVal_Real_Scalar = R

  return
end function MaxVal_Real_Scalar
```

B.2.5 MinVal_Real_Scalar Procedure

The main documentation of the MinVal_Real_Scalar Procedure in § 7.2.5 on page 45 contains additional explanation of this code listing.

```
function MinVal_Real_Scalar (R)

  ! Input variable.

  type(real), intent(in) :: R

  ! Output variable.

  type(real) :: MinVal_Real_Scalar

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ! MinVal_Real_Scalar is equal to R.

    MinVal_Real_Scalar = R

    return
  end function MinVal_Real_Scalar
```

B.2.6 SUM_Real_Scalar Procedure

The main documentation of the SUM_Real_Scalar Procedure in § 7.2.6 on page 45 contains additional explanation of this code listing.

```
  function SUM_Real_Scalar (R)

    ! Input variable.

    type(real), intent(in) :: R

    ! Output variable.

    type(real) :: SUM_Real_Scalar

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! SUM_Real_Scalar is equal to R.

    SUM_Real_Scalar = R

    return
  end function SUM_Real_Scalar
```

B.2.7 VeryClose_Real Procedure

The main documentation of the VeryClose_Real Procedure in § 7.2.7 on page 46 contains additional explanation of this code listing.

```
  define([VERY_CLOSE_ROUTINE],[
    pushdef([DIM], [$1])
    pushdef([VeryClose_Real_DIM], expand(VeryClose_Real_DIM))

    function VeryClose_Real_DIM (X, Y) result(VeryClose)

      ! Use association for numbers.

      use Caesar_Numbers_Module, only: two, ten

      ! Input variables.

      type(real,DIM,np), intent(in) :: X, Y         ! Variables to be checked.
```

```
      ! Output variable.

      type(logical) :: VeryClose                    ! Result of check.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(ALL(SHAPE(X)==SHAPE(Y)),5)  ! X and Y are conformable.

      ! VeryClose is true if X and Y are within 10 SPACING's of each other.

      VeryClose = ALL(ABS(X - Y) < ten*SPACING((X+Y)/two))

      ! Verify guarantees -- none.

      return
    end function VeryClose_Real_DIM

    popdef([DIM])
    popdef([VeryClose_Real_DIM])
  ])

  forloop([Dim],[0],[7],[
    VERY_CLOSE_ROUTINE(Dim)
  ])
```

B.2.8 Real Class Unit Test Program

This lightly commented program performs a unit test on the Real Class, which is described in § 7.2 on page 43.

```
module Unit_Test_Module
  use Caesar_Real_Class
  implicit none

contains

  subroutine testreal (R)
    type(real) :: R
    type(logical) :: vs
    write (6,100) 'R = ', R
    vs = Valid_State(R)
    write (6,101) 'Valid_State(R)         ==> ', vs
    100 format (/, a, 1pe15.6)
    101 format (2x, a, l1)
    return
  end subroutine testreal

  subroutine testreal3 (R3)
    type(real,3) :: R3
```

```
    type(logical) :: vs
    write (6,100) 'R3(1,1,1) = ', R3(1,1,1)
    vs = Valid_State(R3)
    write (6,101) 'Valid_State(R3)        ==> ', vs
    100 format (/, a, 1pe15.6)
    101 format (2x, a, l1)
    return
  end subroutine testreal3

end module Unit_Test_Module

program Unit_Test
  use Unit_Test_Module
  use Caesar_Real_Class
  implicit none

  type(real) :: R, R2
  type(real,3) :: R3
  type(real) :: one, zero

  ! Initializations.

  call Initialize (R)
  call Initialize (R2)
  call Initialize (R3, 3, 4, 5)

  ! Parameters are not used here
  ! to fool smart compilers.

  one = 1.d0
  zero = one - one

  ! Real tests.

  ifdef([DIVISION_BY_ZERO],[
    R = one/zero
    call testreal (R)
    R = -one/zero
    call testreal (R)
    R = zero/zero
    call testreal (R)
  ])
  R = zero
  call testreal (R)
  R = one
  call testreal (R)
  R = HUGE(one)
  call testreal (R)
  R = -HUGE(one)
  call testreal (R)

  ! Real multi-dimensional tests.

  R3 = one
```

```
  ifdef([DIVISION_BY_ZERO],[
    R3(1,1,1) = one/zero
    call testreal3 (R3)
    R3(1,1,1) = -one/zero
    call testreal3 (R3)
    R3(1,1,1) = zero/zero
    call testreal3 (R3)
  ])
  R3(1,1,1) = zero
  call testreal3 (R3)
  R3(1,1,1) = one
  call testreal3 (R3)
  R3(1,1,1) = HUGE(one)
  call testreal3 (R3)
  R3(1,1,1) = -HUGE(one)
  call testreal3 (R3)

  ! Real scalar function tests.

  write (6,*)
  write (6,*) 'Real scalar function tests:'
  R = 1.23456789d0
  write (6,*) 'MaxVal(R) = ', MaxVal(R)
  write (6,*) 'MinVal(R) = ', MinVal(R)
  write (6,*) 'SUM(R)    = ', SUM(R)
  R2 = (((R + 1.d0) * 47.d0) - 47.d0) / 47.d0
  if (.not. VeryClose(R, R2)) then
    write (6,*) 'VeryClose Error: '
    write (6,*) '  R  = ', R
    write (6,*) '  R2 = ', R2
  end if

  ! Finalizations.

  call Finalize (R)
  call Finalize (R2)
  call Finalize (R3)

end
```

B.3 Integer Class Code Listing

The main documentation of the Integer Class in § 7.3 on page 46 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
```

```
! Created on: 1/18/99
! CVS Info:   $Id: integer.F90,v 7.2 2006/10/12 18:30:29 hall Exp $

module Caesar_Integer_Class

  ! Global use associations.

  use Caesar_Status_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Valid_State_NP
  public :: MaxVal, MinVal, SUM

  REPLICATE_INTERFACE([Initialize], [Initialize_Integer])

  REPLICATE_INTERFACE([Finalize], [Finalize_Integer])

  REPLICATE_INTERFACE([Valid_State], [Valid_State_Integer_P])
  REPLICATE_INTERFACE([Valid_State_NP], [Valid_State_Integer_NP])

  interface MaxVal
    module procedure MaxVal_Integer_Scalar
  end interface

  interface MinVal
    module procedure MinVal_Integer_Scalar
  end interface

  interface SUM
    module procedure SUM_Integer_Scalar
  end interface

contains
```

The Integer_Class contains the following routines which are listed in separate sections:

Initialize_Integer (§ B.3.1, page 248)

Finalize_Integer (§ B.3.2, page 249)

Valid_State_Integer (§ B.3.3, page 251)

MaxVal_Integer_Scalar (§ B.3.4, page 252)

MinVal_Integer_Scalar (§ B.3.5, page 253)

SUM_Integer_Scalar (§ B.3.6, page 253)

end module Caesar_Integer_Class

B.3.1 Initialize_Integer Procedure

The main documentation of the Initialize_Integer Procedure in § 7.3.1 on page 46 contains additional explanation of this code listing.

```
subroutine Initialize_Integer_0 (I, status)

  ! Use association information.

  use Caesar_Flags_Module, only: initialize_integer_flag

  ! Output variables.

  type(integer), intent(out) :: I     ! Variable to be initialized.
  type(Status_type), intent(out), optional :: status ! Exit status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Initialize to flag value.

  I = initialize_integer_flag

  ! No errors for initialization possible for scalars.

  if (PRESENT(status)) status = 'Success'

  ! Verify guarantees - none.

  return
end subroutine Initialize_Integer_0

define([REPLICATE_ROUTINE],[
  subroutine Initialize_Integer_$1 (I REP_ARGS([dim[]i]), status)

    ! Use association information.

    use Caesar_Flags_Module, only: initialize_integer_flag

    ! Input variables.

    REP_DECLARE([type(integer), intent(in)], [dim[]i]) ! Array dimensions.

    ! Input/Output variable.

    type(integer,$1) :: I            ! Variable to be initialized.

    ! Output variable.

    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variable.
```

```
    type(integer) :: allocate_status ! Allocation Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    ! The association status of a unallocated pointer is officially
    ! undefined according to the Fortran standard. With most compilers,
    ! the status is unassociated.
    ifelse(COMPILER, NAGWare,
      [], [
      VERIFY(.not.ASSOCIATED(I), 0)   ! I starts out unassociated.
    ])

    ! Allocation (for arrays only).

    REP_ALLOCATE([I], [dim[]i], [allocate_status])

    ! Initialize to flag value.

    I = initialize_integer_flag

    ! Verify guarantees and/or set status flag.

    if (PRESENT(status)) then
      WARN_IF(allocate_status /= 0, 3)  ! Allocation error check.
      WARN_IF(.not.ASSOCIATED(I),3)     ! I is now associated.
      if (allocate_status == 0 .and. ASSOCIATED(I)) then
        status = 'Success'
      else
        status = 'Memory Error'
      end if
    else
      VERIFY(allocate_status == 0, 0)  ! Allocation error check.
      VERIFY(ASSOCIATED(I),0)          ! I is now associated.
    end if

    return
  end subroutine Initialize_Integer_$1
])

REPLICATE_ARRAYS
```

B.3.2 Finalize_Integer Procedure

The main documentation of the Finalize_Integer Procedure in § 7.3.2 on page 47 contains additional explanation of this code listing.

```
subroutine Finalize_Integer_0 (I, status)

  ! Use association information.
```

```
    use Caesar_Flags_Module, only: finalize_integer_flag

    ! Input/Output variable.

    type(integer), intent(inout) :: I    ! Variable to be finalized.

    ! Output variable.

    type(Status_type), intent(out), optional :: status  ! Exit status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Finalization.

    I = finalize_integer_flag

    ! No errors for finalization possible for scalars.

    if (PRESENT(status)) status = 'Success'

    ! Verify guarantees - none.

    return
  end subroutine Finalize_Integer_0

  define([REPLICATE_ROUTINE],[
    subroutine Finalize_Integer_$1 (I, status)

      ! Input/Output variable.

      type(integer,$1) :: I              ! Variable to be finalized.

      ! Output variable.

      type(Status_type), intent(out), optional :: status  ! Exit status.

      ! Internal variable.

      type(integer) :: deallocate_status ! Deallocation Status.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(ASSOCIATED(I),0)           ! I should be associated.

      ! Deallocation.

      DEALLOCATE(I, stat=deallocate_status)

      ! Finalization and nullification.
```

```
    NULLIFY(I)

    ! Verify guarantees and/or set status flag.

    if (PRESENT(status)) then
      WARN_IF(deallocate_status /= 0, 3)  ! Deallocation error check.
      WARN_IF(ASSOCIATED(I),3)            ! I is now unassociated.
      if (deallocate_status == 0 .and. .not.ASSOCIATED(I)) then
        status = 'Success'
      else
        status = 'Memory Error'
      end if
    else
      VERIFY(deallocate_status == 0, 0)  ! Deallocation error check.
      VERIFY(.not.ASSOCIATED(I), 0)      ! I is now unassociated.
    end if

    return
  end subroutine Finalize_Integer_$1
])

REPLICATE_ARRAYS
```

B.3.3 Valid_State_Integer Procedure

The main documentation of the Valid_State_Integer Procedure in § 7.3.3 on page 48 contains additional explanation of this code listing.

```
define([REPLICATE_ROUTINE],[
  ifelse(POINTER_TOGGLE, [TRUE], [
    pushdef([TYPE], [integer,$1])
    pushdef([Valid_State_Integer_P_DIM], expand(Valid_State_Integer_P_$1))
    pushdef([POINTER_ONLY], [])
  ],[
    pushdef([TYPE], [integer,$1,np])
    pushdef([Valid_State_Integer_P_DIM], expand(Valid_State_Integer_NP_$1))
    pushdef([POINTER_ONLY], [!])
  ])

  function Valid_State_Integer_P_DIM (I) result(Valid)

    ! Use association information.

    SCALAR_ONLY use Caesar_Flags_Module, only: finalize_integer_flag
    SCALAR_ONLY use Caesar_Logical_Class, only: ALL

    ! Input variable.

    type(TYPE) :: I  ! Variable to be checked.

    ! Output variable.
```

```
    type(logical) :: Valid      ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! First, make sure that the variable has been allocated.

    POINTER_ONLY ARRAY_ONLY Valid = Valid .and. ASSOCIATED(I)
    POINTER_ONLY ARRAY_ONLY if (.not.Valid) return

    ! Make sure the variable has not been finalized.

    SCALAR_ONLY Valid = Valid .and. I /= finalize_integer_flag

    ! Check the top of the range.

    Valid = Valid .and. ALL(I <= HUGE(I))

    ! Note that there is no explicit check for the bottom of the
    ! range, since there is no F90 intrinsic that returns the lowest
    ! negative number that an integer can take.

    return
  end function Valid_State_Integer_P_DIM

  popdef([TYPE])
  popdef([Valid_State_Integer_P_DIM])
  popdef([POINTER_ONLY])
])

define([POINTER_TOGGLE], [TRUE])
REPLICATE

define([POINTER_TOGGLE], [FALSE])
REPLICATE
```

B.3.4 MaxVal_Integer_Scalar Procedure

The main documentation of the MaxVal_Integer_Scalar Procedure in § 7.3.4 on page 48 contains additional explanation of this code listing.

```
function MaxVal_Integer_Scalar (I)

  ! Input variable.

  type(integer), intent(in) :: I

  ! Output variable.
```

```
    type(integer) :: MaxVal_Integer_Scalar

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! MaxVal_Integer_Scalar is equal to I.

    MaxVal_Integer_Scalar = I

    return
  end function MaxVal_Integer_Scalar
```

B.3.5 MinVal_Integer_Scalar Procedure

The main documentation of the MinVal_Integer_Scalar Procedure in § 7.3.5 on page 48 contains additional explanation of this code listing.

```
  function MinVal_Integer_Scalar (I)

    ! Input variable.

    type(integer), intent(in) :: I

    ! Output variable.

    type(integer) :: MinVal_Integer_Scalar

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! MinVal_Integer_Scalar is equal to I.

    MinVal_Integer_Scalar = I

    return
  end function MinVal_Integer_Scalar
```

B.3.6 SUM_Integer_Scalar Procedure

The main documentation of the SUM_Integer_Scalar Procedure in § 7.3.6 on page 49 contains additional explanation of this code listing.

```
  function SUM_Integer_Scalar (I)

    ! Input variable.

    type(integer), intent(in) :: I

    ! Output variable.
```

```
    type(integer) :: SUM_Integer_Scalar

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! SUM_Integer_Scalar is equal to I.

    SUM_Integer_Scalar = I

    return
  end function SUM_Integer_Scalar
```

B.3.7 Integer Class Unit Test Program

This lightly commented program performs a unit test on the Integer Class, which is described in § 7.3 on page 46.

```
module Unit_Test_Module
  use Caesar_Integer_Class
  implicit none

contains

  subroutine testint (I)
    type(integer) :: I
    type(logical) :: vs
    write (6,100) 'I = ', I
    vs = Valid_State(I)
    write (6,101) 'Valid_State(I)         ==> ', vs
    100 format (/, a, i14)
    101 format (2x, a, l1)
    return
  end subroutine testint

  subroutine testint3 (I3)
    type(integer), pointer, dimension(:,:,:) :: I3
    type(logical) :: vs
    write (6,100) 'I3(1,1,1) = ', I3(1,1,1)
    vs = Valid_State(I3)
    write (6,101) 'Valid_State(I3)        ==> ', vs
    100 format (/, a, i14)
    101 format (2x, a, l1)
    return
  end subroutine testint3

end module Unit_Test_Module

program Unit_Test
  use Unit_Test_Module
  use Caesar_Integer_Class
  implicit none

  type(integer) :: I
```

```
type(integer,3) :: I3

! Initializations.

call Initialize (I)
call Initialize (I3, 3, 4, 5)

! Integer tests.

I = 0
call testint (I)
I = HUGE(I)
call testint (I)
I = -HUGE(I)
call testint (I)

I3(1,1,1) = 0
call testint3 (I3)
I3(1,1,1) = HUGE(I3)
call testint3 (I3)
I3(1,1,1) = -HUGE(I3)
call testint3 (I3)

! Note that the compiler figures out the error unless it is given in
! two steps, as follows. Also, note that adding 1 to HUGE wraps to
! a low negative number, as does subtracting 2 from -HUGE.

I = HUGE(I)
I = I+1
call testint (I)
I = -HUGE(I)
I = I-2
call testint (I)

I3(1,1,1) = HUGE(I3)
I3(1,1,1) = I3(1,1,1)+1
call testint3 (I3)
I3(1,1,1) = -HUGE(I3)
I3(1,1,1) = I3(1,1,1)-2
call testint3 (I3)

! The bottom line is that there are really no invalid integers. The
! Valid_State_Integer routine is primarily added for completeness.

! Integer scalar function tests.

I = 123456789
write (6,*) 'MaxVal(I) = ', MaxVal(I)
write (6,*) 'MinVal(I) = ', MinVal(I)
write (6,*) 'SUM(I)    = ', SUM(I)

! Finalizations.

call Finalize (I)
```

```
  call Finalize (I3)

end
```

B.4 Logical Class Code Listing

The main documentation of the Logical Class in § 7.4 on page 49 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 1/11/99
! CVS Info:   $Id: logical.F90,v 7.8 2006/10/12 18:30:29 hall Exp $

module Caesar_Logical_Class

  ! Global use associations.

  use Caesar_Status_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Valid_State_NP
  public :: ALL, ANY, COUNT, Operator(.InInterval.), Operator(.InSet.), &
            Operator(.NotInInterval.), Operator(.NotInSet.)

  REPLICATE_INTERFACE([Initialize], [Initialize_Logical])

  REPLICATE_INTERFACE([Finalize], [Finalize_Logical])

  REPLICATE_INTERFACE([Valid_State], [Valid_State_Logical_P])
  REPLICATE_INTERFACE([Valid_State_NP], [Valid_State_Logical_NP])

  interface ALL
    module procedure ALL_Scalar
  end interface

  interface ANY
    module procedure ANY_Scalar
  end interface

  interface COUNT
```

```
    module procedure COUNT_Scalar
  end interface

  define([OPERATOR_INTERFACE],[
    pushdef([PROCEDURE], [$1])
    pushdef([TYPE], [$2])
    pushdef([DIM], [$3])
    pushdef([PROCEDURE_TYPE_DIM], expand(PROCEDURE_TYPE_DIM))

    interface OPERATOR (.PROCEDURE.)
      module procedure PROCEDURE_TYPE_DIM
    end interface

    popdef([PROCEDURE])
    popdef([TYPE])
    popdef([DIM])
    popdef([PROCEDURE_TYPE_DIM])
  ])

  forloop([Dim],[0],[7],[
    fortext([Type],[Real Integer],[
      fortext([Proc],[InInterval NotInInterval],[
        OPERATOR_INTERFACE(Proc, Type, Dim)
      ])
    ])
  ])

  define([OPERATOR_INTERFACE],[
    pushdef([PROCEDURE], [$1])
    pushdef([TYPE], [$2])
    pushdef([PROCEDURE_TYPE], expand(PROCEDURE_TYPE))

    interface OPERATOR (.PROCEDURE.)
      module procedure PROCEDURE_TYPE
    end interface

    popdef([PROCEDURE])
    popdef([TYPE])
    popdef([PROCEDURE_TYPE])
  ])

  fortext([Type],[Real Integer Character],[
    fortext([Proc],[InSet NotInSet],[
      OPERATOR_INTERFACE(Proc, Type)
    ])
  ])

contains
```

The Logical_Class contains the following routines which are listed in separate sections:

Initialize_Logical (§ B.4.1, page 258)

Finalize_Logical (§ B.4.2, page 260)

end module Caesar_Logical_Class

B.4.1 Initialize_Logical Procedure

The main documentation of the Initialize_Logical Procedure in § 7.4.1 on page 49 contains additional explanation of this code listing.

```
subroutine Initialize_Logical_0 (L, status)

  ! Use association information.

  use Caesar_Flags_Module, only: initialize_logical_flag

  ! Output variables.

  type(logical), intent(out) :: L      ! Variable to be initialized.
  type(Status_type), intent(out), optional :: status  ! Exit status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Initialize.

  L = initialize_logical_flag

  ! No errors for initialization possible for scalars.

  if (PRESENT(status)) status = 'Success'

  ! Verify guarantees - none.

  return
end subroutine Initialize_Logical_0

define([REPLICATE_ROUTINE],[
  subroutine Initialize_Logical_$1 (L REP_ARGS([dim[]i]), status)

    ! Use association information.
```

```
use Caesar_Flags_Module, only: initialize_logical_flag

! Input variables.

REP_DECLARE([type(integer), intent(in)], [dim[]i])  ! Array dimensions.

! Input/Output variable.

type(logical,$1) :: L            ! Variable to be initialized.

! Output variable.

type(Status_type), intent(out), optional :: status  ! Exit status.

! Internal variable.

type(integer) :: allocate_status ! Allocation Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

! The association status of a unallocated pointer is officially
! undefined according to the Fortran standard. With most compilers,
! the status is unassociated.
ifelse(COMPILER, NAGWare,
  [], [
  VERIFY(.not.ASSOCIATED(L), 0)   ! L starts out unassociated.
])

! Allocation (for arrays only).

REP_ALLOCATE([L], [dim[]i], [allocate_status])

! Initialize.

L = initialize_logical_flag

! Verify guarantees and/or set status flag.

if (PRESENT(status)) then
  WARN_IF(allocate_status /= 0, 3)  ! Allocation error check.
  WARN_IF(.not.ASSOCIATED(L),3)     ! L is now associated.
  if (allocate_status == 0 .and. ASSOCIATED(L)) then
    status = 'Success'
  else
    status = 'Memory Error'
  end if
else
  VERIFY(allocate_status == 0, 0)  ! Allocation error check.
  VERIFY(ASSOCIATED(L),0)          ! L is now associated.
end if
```

```
    return
  end subroutine Initialize_Logical_$1
])

REPLICATE_ARRAYS
```

B.4.2 Finalize_Logical Procedure

The main documentation of the Finalize_Logical Procedure in § 7.4.2 on page 50 contains additional explanation of this code listing.

```
subroutine Finalize_Logical_0 (L, status)

  ! Use association information.

  use Caesar_Flags_Module, only: finalize_logical_flag

  ! Input/Output variable.

  type(logical), intent(inout) :: L    ! Variable to be finalized.

  ! Output variable.

  type(Status_type), intent(out), optional :: status  ! Exit status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Finalization.

  L = finalize_logical_flag

  ! No errors for finalization possible for scalars.

  if (PRESENT(status)) status = 'Success'

  ! Verify guarantees - none.

  return
end subroutine Finalize_Logical_0

define([REPLICATE_ROUTINE],[
  subroutine Finalize_Logical_$1 (L, status)

    ! Input/Output variable.

    type(logical,$1) :: L              ! Variable to be finalized.

    ! Output variable.

    type(Status_type), intent(out), optional :: status  ! Exit status.
```

```
    ! Internal variable.

    type(integer) :: deallocate_status ! Deallocation Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(ASSOCIATED(L),0)           ! L should be associated.

    ! Deallocation.

    DEALLOCATE(L, stat=deallocate_status)

    ! Finalization and nullification.

    NULLIFY(L)

    ! Verify guarantees and/or set status flag.

    if (PRESENT(status)) then
      WARN_IF(deallocate_status /= 0, 3)  ! Deallocation error check.
      WARN_IF(ASSOCIATED(L),3)            ! L is now unassociated.
      if (deallocate_status == 0 .and. .not.ASSOCIATED(L)) then
        status = 'Success'
      else
        status = 'Memory Error'
      end if
    else
      VERIFY(deallocate_status == 0, 0)  ! Deallocation error check.
      VERIFY(.not.ASSOCIATED(L), 0)      ! L is now unassociated.
    end if

    return
  end subroutine Finalize_Logical_$1
])

REPLICATE_ARRAYS
```

B.4.3 Valid_State_Logical Procedure

The main documentation of the Valid_State_Logical Procedure in § 7.4.3 on page 51 contains additional explanation of this code listing.

```
define([REPLICATE_ROUTINE],[
  ifelse(POINTER_TOGGLE, [TRUE], [
    pushdef([TYPE], [logical,$1])
    pushdef([Valid_State_Logical_P_DIM], expand(Valid_State_Logical_P_$1))
    pushdef([POINTER_ONLY], [])
  ],[
    pushdef([TYPE], [logical,$1,np])
```

```
    pushdef([Valid_State_Logical_P_DIM], expand(Valid_State_Logical_NP_$1))
    pushdef([POINTER_ONLY], [!])
  ])

  function Valid_State_Logical_P_DIM (L) result(Valid)

    ! Input variable.

    type(TYPE) :: L              ! Variable to be checked.

    ! Output variable.

    type(logical) :: Valid       ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! First, make sure that the variable has been allocated.

    POINTER_ONLY ARRAY_ONLY Valid = Valid .and. ASSOCIATED(L)
    POINTER_ONLY ARRAY_ONLY if (.not.Valid) return

    ! All logicals should be valid, so this tautology should always work.

    Valid = Valid .and. ALL(L .or. .not. L)

    return
  end function Valid_State_Logical_P_DIM

  popdef([TYPE])
  popdef([Valid_State_Logical_P_DIM])
  popdef([POINTER_ONLY])
])

define([POINTER_TOGGLE], [TRUE])
REPLICATE

define([POINTER_TOGGLE], [FALSE])
REPLICATE
```

B.4.4 ALL_Scalar Procedure

The main documentation of the ALL_Scalar Procedure in § 7.4.4 on page 51 contains additional explanation of this code listing.

```
function ALL_Scalar (L)

  ! Input variable.
```

```
    type(logical), intent(in) :: L

    ! Output variable.

    type(logical) :: ALL_Scalar

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! ALL_Scalar is true iff L is true.

    ALL_Scalar = L

    return
  end function ALL_Scalar
```

B.4.5 ANY_Scalar Procedure

The main documentation of the ANY_Scalar Procedure in § 7.4.5 on page 51 contains additional explanation of this code listing.

```
  function ANY_Scalar (L)

    ! Input variable.

    type(logical), intent(in) :: L

    ! Output variable.

    type(logical) :: ANY_Scalar

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! ANY_Scalar is true iff L is true.

    ANY_Scalar = L

    return
  end function ANY_Scalar
```

B.4.6 COUNT_Scalar Procedure

The main documentation of the COUNT_Scalar Procedure in § 7.4.6 on page 52 contains additional explanation of this code listing.

```
  function COUNT_Scalar (L)

    ! Input variable.

    type(logical), intent(in) :: L
```

```
    ! Output variable.

    type(integer) :: COUNT_Scalar

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! COUNT_Scalar is equal to the number of trues in L,
    ! so it is 1 if L is true, 0 if L is false.

    if (L) then
      COUNT_Scalar = 1
    else
      COUNT_Scalar = 0
    end if

    return
  end function COUNT_Scalar
```

B.4.7 InInterval Procedure

The main documentation of the InInterval Procedure in § 7.4.7 on page 52 contains additional explanation of this code listing.

```
  define([IN_INTERVAL_ROUTINE],[
    pushdef([TYPE], [$1])
    pushdef([DIM], [$2])
    pushdef([InInterval_TYPE_DIM], expand(InInterval_TYPE_DIM))

    function InInterval_TYPE_DIM (X, Interval) result(InInterval)

      ! Input variables.

      type(TYPE,DIM,np), intent(in) :: X              ! Variable to be checked.
      type(TYPE), dimension(2), intent(in) :: Interval ! Interval to check.

      ! Output variable.

      type(logical) :: InInterval                      ! Result of check.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Interval(1) <= Interval(2),7)   ! Interval should be well-formed.

      ! InInterval is true if X is in the interval.

      InInterval = ALL(X >= Interval(1) .and. &
                       X <= Interval(2))

      ! Verify guarantees -- none.
```

```
    return
  end function InInterval_TYPE_DIM

  popdef([TYPE])
  popdef([DIM])
  popdef([InInterval_TYPE_DIM])
])

forloop([Dim],[0],[7],[
  fortext([Type],[real integer],[
    IN_INTERVAL_ROUTINE(Type, Dim)
  ])
])
```

B.4.8 InSet Procedure

The main documentation of the InSet Procedure in § 7.4.8 on page 53 contains additional explanation of this code listing.

```
define([IN_SET_ROUTINE],[
  ifelse($1, [character], [
    pushdef([TYPE], [$1,*])
    pushdef([InSet_TYPE], expand(InSet_$1))
  ],[
    pushdef([TYPE], [$1])
    pushdef([InSet_TYPE], expand(InSet_TYPE))
  ])

  function InSet_TYPE (X, Set) result(InSet)

    ! Input variables.

    type(TYPE), intent(in) :: X                 ! Variable to be checked.
    type(TYPE), dimension(:), intent(in) :: Set ! The set to check.

    ! Output variable.

    type(logical) :: InSet                      ! Result of check.

    ! Internal variable.

    type(integer) :: element                    ! Element loop counter.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! InSet is true if X is in the set.

    InSet = .false.
    do element = 1, SIZE(Set)
```

```
      InSet = InSet .or. X == Set(element)
    end do

    ! Verify guarantees -- none.

    return
  end function InSet_TYPE

  popdef([TYPE])
  popdef([InSet_TYPE])
])

fortext([Type],[real integer character],[
  IN_SET_ROUTINE(Type)
])
```

B.4.9 NotInInterval Procedure

The main documentation of the NotInInterval Procedure in § 7.4.9 on page 53 contains additional explanation of this code listing.

```
define([NOT_IN_INTERVAL_ROUTINE],[
  pushdef([TYPE], [$1])
  pushdef([DIM], [$2])
  pushdef([NotInInterval_TYPE_DIM], expand(NotInInterval_TYPE_DIM))

  function NotInInterval_TYPE_DIM (X, Interval) result(NotInInterval)

    ! Input variables.

    type(TYPE,DIM,np), intent(in) :: X              ! Variable to be checked.
    type(TYPE), dimension(2), intent(in) :: Interval ! Interval to check.

    ! Output variable.

    type(logical) :: NotInInterval                   ! Result of check.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Interval(1) <= Interval(2),7)   ! Interval should be well-formed.

    ! NotInInterval is true if X is not in the interval.

    NotInInterval = ALL(X < Interval(1) .or. &
                        X > Interval(2))

    ! Verify guarantees -- none.

    return
  end function NotInInterval_TYPE_DIM
```

```
    popdef([TYPE])
    popdef([DIM])
    popdef([NotInInterval_TYPE_DIM])
  ])

  forloop([Dim],[0],[7],[
    fortext([Type],[real integer],[
      NOT_IN_INTERVAL_ROUTINE(Type, Dim)
    ])
  ])
```

B.4.10 NotInSet Procedure

The main documentation of the NotInSet Procedure in § 7.4.10 on page 54 contains additional explanation of this code listing.

```
  define([NOT_IN_SET_ROUTINE],[
    ifelse($1, [character], [
      pushdef([TYPE], [$1,*])
      pushdef([NotInSet_TYPE], expand(NotInSet_$1))
    ],[
      pushdef([TYPE], [$1])
      pushdef([NotInSet_TYPE], expand(NotInSet_TYPE))
    ])

    function NotInSet_TYPE (X, Set) result(NotInSet)

      ! Input variables.

      type(TYPE), intent(in) :: X                 ! Variable to be checked.
      type(TYPE), dimension(:), intent(in) :: Set ! The set to check.

      ! Output variable.

      type(logical) :: NotInSet                   ! Result of check.

      ! Internal variable.

      type(integer) :: element                    ! Element loop counter.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements - none.

      ! NotInSet is true if X is not in the set.

      NotInSet = .true.
      do element = 1, SIZE(Set)
        NotInSet = NotInSet .and. X /= Set(element)
      end do
```

```
      ! Verify guarantees -- none.

      return
    end function NotInSet_TYPE

    popdef([TYPE])
    popdef([NotInSet_TYPE])
  ])

  fortext([Type],[real integer character],[
    NOT_IN_SET_ROUTINE(Type)
  ])
```

B.4.11 Logical Class Unit Test Program

This lightly commented program performs a unit test on the Logical Class, which is described in § 7.4 on page 49.

```
program Unit_Test
  use Caesar_Logical_Class
  implicit none

  type(logical) :: L
  type(logical,3) :: L3

  ! Initialize logicals.

  call Initialize (L)
  call Initialize (L3, 3, 4, 5)

  ! Logical Valid_State tests.

  L = .false.
  write (6,*) 'Valid_State L =', Valid_State(L)
  L = .true.
  write (6,*) 'Valid_State L =', Valid_State(L)
  L3 = .false.
  write (6,*) 'Valid_State L3 =', Valid_State(L3)
  L3 = .true.
  write (6,*) 'Valid_State L3 =', Valid_State(L3)

  ! Logical scalar function tests.

  L = .false.
  write (6,*) 'COUNT(L) =', COUNT(L)
  write (6,*) 'ALL(L) =', ALL(L)
  write (6,*) 'ANY(L) =', ANY(L)
  L = .true.
  write (6,*) 'COUNT(L) =', COUNT(L)
  write (6,*) 'ALL(L) =', ALL(L)
  write (6,*) 'ANY(L) =', ANY(L)
```

```
  ! Logical interval tests.

  write (6,*) 'Following intervals should use brackets, '
  write (6,*) 'but they are not available in F90.'
  write (6,*) 'Is 1 in (1,99)?:', 1 .InInterval. (/1, 99/)
  write (6,*) 'Is 5 in (3,7)?:', 5 .InInterval. (/3, 7/)
  write (6,*) 'Is (6,3,4) in (3,7)?:', &
              (/6, 3, 4/) .InInterval. (/3, 7/)
  write (6,*) 'Is (6,3,4,8) in (3,7)?:', &
              (/6, 3, 4, 8/) .InInterval. (/3, 7/)
  write (6,*) 'Is 1.0 in (1.0,99.0)?:', 1.d0 .InInterval. (/1.d0, 99.d0/)
  write (6,*) 'Is 3.14159 in (2.3,10.4)?:', &
              3.14159d0 .InInterval. (/2.3d0, 10.4d0/)
  write (6,*) 'Is (3.14159,4.2,5.8) in (2.3,10.4)?:', &
              (/3.14159d0, 4.2d0, 5.8d0/) .InInterval. (/2.3d0, 10.4d0/)
  write (6,*) 'Is (3.14159,4.2,5.8,14.0) in (2.3,10.4)?:', &
              (/3.14159d0, 4.2d0, 5.8d0, 14.d0/) .InInterval. (/2.3d0, 10.4d0/)

  write (6,*) 'Is 1 not in (1,99)?:', 1 .NotInInterval. (/1, 99/)
  write (6,*) 'Is 1 not in (3,7)?:', 1 .NotInInterval. (/3, 7/)
  write (6,*) 'Is (6,3,4) not in (3,7)?:', &
              (/6, 3, 4/) .NotInInterval. (/3, 7/)
  write (6,*) 'Is (16,-3,14,8) not in (3,7)?:', &
              (/16, -3, 14, 8/) .NotInInterval. (/3, 7/)
  write (6,*) 'Is 1.0 not in (1.0,99.0)?:', 1.d0 .NotInInterval. (/1.d0, 99.d0/)
  write (6,*) 'Is 3.14159 not in (2.3,10.4)?:', &
              3.14159d0 .NotInInterval. (/2.3d0, 10.4d0/)
  write (6,*) 'Is (3.14159,4.2,5.8) not in (2.3,10.4)?:', &
              (/3.14159d0, 4.2d0, 5.8d0/) .NotInInterval. (/2.3d0, 10.4d0/)
  write (6,*) 'Is (-3.14159,14.2,-5.8,14.0) not in (2.3,10.4)?:', &
              (/-3.14159d0, 14.2d0, -5.8d0, 14.d0/) &
              .NotInInterval. (/2.3d0, 10.4d0/)

  ! Finalize logicals.

  call Finalize (L)
  call Finalize (L3)

end
```

B.5 Character Class Code Listing

The main documentation of the Character Class in § 7.5 on page 54 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
```

```
! Created on: 1/18/99
! CVS Info:   $Id: character.F90,v 6.10 2006/10/12 18:30:29 hall Exp $

module Caesar_Character_Class

  ! Global use associations.

  use Caesar_Status_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Valid_State_NP

  REPLICATE_INTERFACE([Initialize], [Initialize_Character])

  REPLICATE_INTERFACE([Finalize], [Finalize_Character])

  REPLICATE_INTERFACE([Valid_State], [Valid_State_Character_P])
  REPLICATE_INTERFACE([Valid_State_NP], [Valid_State_Character_NP])

contains
```

The Character_Class contains the following routines which are listed in separate sections:

Initialize_Character (§ B.5.1, page 270)

Finalize_Character (§ B.5.2, page 272)

Valid_State_Character (§ B.5.3, page 274)

end module Caesar_Character_Class

B.5.1 Initialize_Character Procedure

The main documentation of the Initialize_Character Procedure in § 7.5.1 on page 54 contains additional explanation of this code listing.

```
  subroutine Initialize_Character_0 (C, status)

    ! Use association information.

    use Caesar_Flags_Module, only: initialize_character_flag

    ! Output variables.

    type(character,*), intent(out) :: C  ! Variable to be initialized.
    type(Status_type), intent(out), optional :: status  ! Exit status.
```

```
  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Initialize to flag value.

  C = initialize_character_flag

  ! No errors for initialization possible for scalars.

  if (PRESENT(status)) status = 'Success'

  ! Verify guarantees - none.

  return
end subroutine Initialize_Character_0

define([REPLICATE_ROUTINE],[
  subroutine Initialize_Character_$1 (C REP_ARGS([dim[]i]), status)

    ! Use association information.

    use Caesar_Flags_Module, only: initialize_character_flag

    ! Input variable.

    REP_DECLARE([type(integer), intent(in)], [dim[]i]) ! Array dimensions.

    ! Input/Output variable.

    type(character,*,$1) :: C            ! Variable to be initialized.

    ! Output variable.

    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variable.

    type(integer) :: allocate_status     ! Allocation Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    ! The association status of a unallocated pointer is officially
    ! undefined according to the Fortran standard. With most compilers,
    ! the status is unassociated.
    ifelse(COMPILER, NAGWare,
      [], [
      VERIFY(.not.ASSOCIATED(C), 0)   ! C starts out unassociated.
    ])

    ! Allocation (for arrays only).
```

```
    REP_ALLOCATE([C], [dim[]i], [allocate_status])

    ! Initialize to flag value.

    C = initialize_character_flag

    ! Verify guarantees and/or set status flag.

    if (PRESENT(status)) then
      WARN_IF(allocate_status /= 0, 3)  ! Allocation error check.
      WARN_IF(.not.ASSOCIATED(C),3)     ! C is now associated.
      if (allocate_status == 0 .and. ASSOCIATED(C)) then
        status = 'Success'
      else
        status = 'Memory Error'
      end if
    else
      VERIFY(allocate_status == 0, 0)  ! Allocation error check.
      VERIFY(ASSOCIATED(C),0)          ! C is now associated.
    end if

    return
  end subroutine Initialize_Character_$1
])

REPLICATE_ARRAYS
```

B.5.2 Finalize_Character Procedure

The main documentation of the Finalize_Character Procedure in § 7.5.2 on page 55 contains additional explanation of this code listing.

```
subroutine Finalize_Character_0 (C, status)

  ! Use association information.

  use Caesar_Flags_Module, only: finalize_character_flag

  ! Input/Output variable.

  type(character,*), intent(inout) :: C    ! Variable to be finalized.

  ! Output variable.

  type(Status_type), intent(out), optional :: status  ! Exit status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Finalization.
```

```
    C = finalize_character_flag

    ! No errors for finalization possible for scalars.

    if (PRESENT(status)) status = 'Success'

    ! Verify guarantees - none.

    return
  end subroutine Finalize_Character_0

  define([REPLICATE_ROUTINE],[
    subroutine Finalize_Character_$1 (C, status)

      ! Input/Output variable.

      type(character,*,$1) :: C          ! Variable to be finalized.

      ! Output variable.

      type(Status_type), intent(out), optional :: status  ! Exit status.

      ! Internal variable.

      type(integer) :: deallocate_status ! Deallocation Status.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(ASSOCIATED(C),0)           ! C should be associated.

      ! Deallocation.

      DEALLOCATE(C, stat=deallocate_status)

      ! Finalization and nullification.

      NULLIFY(C)

      ! Verify guarantees and/or set status flag.

      if (PRESENT(status)) then
        WARN_IF(deallocate_status /= 0, 3)  ! Deallocation error check.
        WARN_IF(ASSOCIATED(C),3)            ! C is now unassociated.
        if (deallocate_status == 0 .and. .not.ASSOCIATED(C)) then
          status = 'Success'
        else
          status = 'Memory Error'
        end if
      else
        VERIFY(deallocate_status == 0, 0)  ! Deallocation error check.
        VERIFY(.not.ASSOCIATED(C), 0)      ! C is now unassociated.
      end if
```

```
    return
  end subroutine Finalize_Character_$1
])

REPLICATE_ARRAYS
```

B.5.3 Valid_State_Character Procedure

The main documentation of the Valid_State_Character Procedure in § 7.5.3 on page 55 contains additional explanation of this code listing.

```
define([REPLICATE_ROUTINE],[
  ifelse(POINTER_TOGGLE, [TRUE], [
    pushdef([TYPE], [character,*,$1])
    pushdef([Valid_State_Character_P_DIM], expand(Valid_State_Character_P_$1))
    pushdef([POINTER_ONLY], [])
    pushdef([NONPOINTER_ONLY], [!])
  ],[
    pushdef([TYPE], [character,*,$1,np])
    pushdef([Valid_State_Character_P_DIM], expand(Valid_State_Character_NP_$1))
    pushdef([POINTER_ONLY], [!])
    pushdef([NONPOINTER_ONLY], [])
  ])

  function Valid_State_Character_P_DIM (C) result(Valid)

    ! Use association information.

    SCALAR_ONLY use Caesar_Flags_Module, only: finalize_character_flag

    ! Input variable.

    type(TYPE) :: C              ! Variable to be checked.

    ! Output variable.

    type(logical) :: Valid       ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! First, make sure that the variable has been allocated.

    POINTER_ONLY ARRAY_ONLY Valid = Valid .and. ASSOCIATED(C)
    POINTER_ONLY ARRAY_ONLY if (.not.Valid) return

    ! Make sure the variable has not been finalized.
```

```
      SCALAR_ONLY Valid = Valid .and. C /= finalize_character_flag

      ! Quiet compiler warnings by making sure C is always referenced.
      ! Note the 'or' which means that this test has no effect.

      !NONPOINTER_ONLY ARRAY_ONLY Valid = Valid .or. LEN(C) /= 0

      return
    end function Valid_State_Character_P_DIM

    popdef([TYPE])
    popdef([Valid_State_Character_P_DIM])
    popdef([POINTER_ONLY])
    popdef([NONPOINTER_ONLY])
  ])

  define([POINTER_TOGGLE], [TRUE])
  REPLICATE

  define([POINTER_TOGGLE], [FALSE])
  REPLICATE
```

B.5.4 Character Class Unit Test Program

This lightly commented program performs a unit test on the Character Class, which is described in § 7.5 on page 54.

```
program Unit_Test

  use Caesar_Character_Class
  implicit none

  type(character,10) :: C
  type(character,20), pointer, dimension(:,:,:) :: C3

  ! Initializations.

  call Initialize (C)
  call Initialize (C3, 3, 4, 5)

  ! Character tests.

  VERIFY(Valid_State(C),0)
  VERIFY(Valid_State(C3),0)
  write (6,*) 'C         = ', C
  write (6,*) 'C3(1,1,1) = ', C3(1,1,1)

  C = 'Test Value'
  C3(1,1,1) = 'Test Value'

  VERIFY(Valid_State(C),0)
  VERIFY(Valid_State(C3),0)
```

```
  write (6,*) 'C         = ', C
  write (6,*) 'C3(1,1,1) = ', C3(1,1,1)

  ! The bottom line is that there are really no invalid characters. The
  ! Valid_State_Character routine is primarily added for completeness.

  ! Finalizations.

  call Finalize (C)
  call Finalize (C3)

  ! Output scalar value again.

  write (6,*) 'C         = ', C

  ! Should be invalid.

  write(6,*) 'Valid_State(C) =', Valid_State(C)

end
```

Appendix C

Utilities Module Code Listing

The main documentation of the Utilities Module in chapter 8 on page 57 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 04/19/01
! CVS Info:   $Id: utilities.F90,v 1.3 2007/10/10 22:09:30 hall Exp $

module Caesar_Utilities_Module

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_F2003_Utils_Module
  use Caesar_Shell_Utils_Module
  use Caesar_Text_Utils_Module

  ! Start up with everything untyped and public.
  ! Note: this module contains no private information.

  implicit none
  public

end module Caesar_Utilities_Module
```

C.1 F2003_Utils Module Code Listing

The main documentation of the F2003_Utils Module in § 8.1 on page 57 contains additional explanation of this code listing.

```
!
```

```
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 10/20/06
! CVS Info:   $Id: f2003_utils.F90,v 1.4 2008/09/29 22:13:53 hall Exp $

module Caesar_F2003_Utils_Module

  ! Global use associations.

  use Caesar_Intrinsics_Module

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Command_Argument_Count, Get_Command_Argument

  interface Command_Argument_Count
    module procedure Command_Argument_Count_F2003
  end interface

  interface Get_Command_Argument
    module procedure Get_Command_Argument_F2003
  end interface

contains
```

The F2003_Utils Module contains the following routines which are listed in separate sections:

Command_Argument_Count_F2003 (§ C.1.1, page 278)

Get_Command_Argument_F2003 (§ C.1.2, page 279)

end module Caesar_F2003_Utils_Module

C.1.1 Command_Argument_Count_F2003 Procedure

The main documentation of the Command_Argument_Count_F2003 Procedure in § 8.1.1 on page 57 contains additional explanation of this code listing.

```
  function Command_Argument_Count_F2003 () result(Command_Argument_Count)

    ! Local use associations.

    ifelse(COMPILER, NAGWare, [
```

```
    use F90_Unix_Env, only: IARGC
  ])

  ! Output variable.

  type(integer) :: Command_Argument_Count ! The number of command arguments.

  ! Internal variable.

  ifelse(COMPILER, NAGWare,
    [], [
    type(integer) :: IARGC                ! Fortran/C intrinsic function.
  ])

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Set command_argument_count.

  Command_Argument_Count = IARGC()

  ! Verify guarantees.

  VERIFY(Command_Argument_Count > -1,5)  ! Command_Argument_Count is valid.

  return
end function Command_Argument_Count_F2003
```

C.1.2 Get_Command_Argument_F2003 Procedure

The main documentation of the Get_Command_Argument_F2003 Procedure in § 8.1.2 on page 58 contains additional explanation of this code listing.

```
subroutine Get_Command_Argument_F2003 (Number, Argument)

  ! Local use associations.

  ifelse(COMPILER, NAGWare, [
    use F90_Unix_Env, only: GETARG
  ])

  ! Input variables.

  type(integer), intent(in) :: Number        ! The number of the argument.

  ! Output variables.

  type(character,*), intent(out) :: Argument ! The argument.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ! Verify requirements.

    ! Number is in the correct range.
    VERIFY(Number.InInterval.(/0,COMMAND_ARGUMENT_COUNT()/),5)

    ! Get the argument.

    call GETARG (Number, Argument)

    ! Verify guarantees - none.

    return
  end subroutine Get_Command_Argument_F2003
```

C.1.3 F2003_Utils Module Unit Test Program

This lightly commented program performs a unit test on the F2003_Utils Module, which is described in § 8.1 on page 57.

```
program Unit_Test

  use Caesar_Intrinsics_Module
  use Caesar_F2003_Utils_Module
  implicit none

  type(integer) :: i
  type(character,80) :: Argument

  ! Testing statements.

  write (6,*) ' '
  write (6,100) 'Command_Argument_Count =  ', COMMAND_ARGUMENT_COUNT()
  do i = 0, COMMAND_ARGUMENT_COUNT()
    call GET_COMMAND_ARGUMENT (i, Argument)
    write (6,100) 'Argument ', i, ' =              ', TRIM(Argument)
  end do

100 format (' ', a, i2, :, a, a)

end
```

C.2 Shell_Utils Module Code Listing

The main documentation of the Shell_Utils Module in § 8.2 on page 58 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D409, LANL
```

```
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 04/19/01
! CVS Info:   $Id: shell_utils.F90,v 1.5 2005/01/27 16:42:40 hall Exp $

module Caesar_Shell_Utils_Module

  ! Global use associations.

  use Caesar_Intrinsics_Module

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Basename, Dirname

  interface Basename
    module procedure Basename_Shell_Utils
  end interface

  interface Dirname
    module procedure Dirname_Shell_Utils
  end interface

contains
```

The Shell_Utils Module contains the following routines which are listed in separate sections:

Basename_Shell_Utils (§ C.2.1, page 281)

Dirname_Shell_Utils (§ C.2.2, page 283)

end module Caesar_Shell_Utils_Module

C.2.1 Basename_Shell_Utils Procedure

The main documentation of the Basename_Shell_Utils Procedure in § 8.2.1 on page 59 contains additional explanation of this code listing.

```
  function Basename_Shell_Utils (Filename, Suffix_Strip) result(Basename)

    ! Input variables.

    type(character,*), intent(in) :: Filename           ! Filename.
    type(logical), intent(in), optional :: Suffix_Strip ! Suffix strip toggle.
```

```
    ! Output variables.

    type(character,255) :: Basename ! The basename of the filename.

    ! Internal variables.

    integer :: basename_left        ! Left extent of the basename.
    integer :: basename_right       ! Right extent of the basename.
    logical :: A_Suffix_Strip       ! Actual suffix strip toggle.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Filename),5)          ! Filename is valid.
    VERIFY(LEN_TRIM(Filename) /= 0,5)        ! Filename is non-null.

    ! Set suffix strip toggle.

    if (PRESENT(Suffix_Strip)) then
      A_Suffix_Strip = Suffix_Strip
    else
      A_Suffix_Strip = .true.
    end if

    ! Determine first character in basename.

    basename_left = MAX(1, INDEX(Filename, '/', .true.) + 1)

    ! Determine final character in basename.

    if (A_Suffix_Strip) then
      basename_right = INDEX(Filename, '.', .true.) - 1
      if (basename_right == -1 .or. basename_right < basename_left) then
        basename_right = LEN_TRIM(Filename)
      end if
    else
      basename_right = LEN_TRIM(Filename)
    end if

    ! Set basename.

    Basename = Filename(basename_left:basename_right)

    ! Verify guarantees.

    VERIFY(Valid_State(Basename),5)  ! Basename is valid.

    return
  end function Basename_Shell_Utils
```

C.2.2 Dirname_Shell_Utils Procedure

The main documentation of the Dirname_Shell_Utils Procedure in § 8.2.2 on page 59 contains additional explanation of this code listing.

```
function Dirname_Shell_Utils (Filename) result(Dirname)

  ! Input variables.

  type(character,*), intent(in) :: Filename          ! Filename.

  ! Output variables.

  type(character,255) :: Dirname ! The dirname of the filename.

  ! Internal variables.

  integer :: dirname_right      ! Right extent of the dirname.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Filename),5)          ! Filename is valid.
  VERIFY(LEN_TRIM(Filename) /= 0,5)        ! Filename is non-null.

  ! Determine final character in dirname.

  dirname_right = INDEX(Filename, '/', .true.) - 1

  ! Set dirname.

  select case (dirname_right)
  case (0)
    Dirname = '/'
  case (-1)
    Dirname = '.'
  case default
    Dirname = Filename(1:dirname_right)
  end select

  ! Verify guarantees.

  VERIFY(Valid_State(Dirname),5)  ! Dirname is valid.

  return
end function Dirname_Shell_Utils
```

C.2.3 Shell_Utils Module Unit Test Program

This lightly commented program performs a unit test on the Shell_Utils Module, which is described in § 8.2 on page 58.

```
program Unit_Test

  use Caesar_Intrinsics_Module
  use Caesar_Shell_Utils_Module
  implicit none

  type(character,80) :: Filename

  ! Testing statements.

  Filename = '/one/two/three/four/five/six.1.2.3'
  write (6,*) ' '
  write (6,*) 'Filename =           ', TRIM(Filename)
  write (6,*) 'Basename =           ', TRIM(Basename(Filename))
  write (6,*) 'Basename, w/suffix = ', TRIM(Basename(Filename,.false.))
  write (6,*) 'Dirname =            ', TRIM(Dirname(Filename))
  write (6,*) 'Reconstructed path = ', &
             TRIM(Dirname(Filename))//'/'//TRIM(Basename(Filename,.false.))

  Filename = '/one'
  write (6,*) ' '
  write (6,*) 'Filename =           ', TRIM(Filename)
  write (6,*) 'Basename =           ', TRIM(Basename(Filename))
  write (6,*) 'Basename, w/suffix = ', TRIM(Basename(Filename,.false.))
  write (6,*) 'Dirname =            ', TRIM(Dirname(Filename))
  write (6,*) 'Reconstructed path = ', &
             TRIM(Dirname(Filename))//'/'//TRIM(Basename(Filename,.false.))

  Filename = '/one/two.three/four.five'
  write (6,*) ' '
  write (6,*) 'Filename =           ', TRIM(Filename)
  write (6,*) 'Basename =           ', TRIM(Basename(Filename))
  write (6,*) 'Basename, w/suffix = ', TRIM(Basename(Filename,.false.))
  write (6,*) 'Dirname =            ', TRIM(Dirname(Filename))
  write (6,*) 'Reconstructed path = ', &
             TRIM(Dirname(Filename))//'/'//TRIM(Basename(Filename,.false.))

  Filename = 'one'
  write (6,*) ' '
  write (6,*) 'Filename =           ', TRIM(Filename)
  write (6,*) 'Basename =           ', TRIM(Basename(Filename))
  write (6,*) 'Basename, w/suffix = ', TRIM(Basename(Filename,.false.))
  write (6,*) 'Dirname =            ', TRIM(Dirname(Filename))
  write (6,*) 'Reconstructed path = ', &
             TRIM(Dirname(Filename))//'/'//TRIM(Basename(Filename,.false.))

  Filename = './one'
  write (6,*) ' '
  write (6,*) 'Filename =           ', TRIM(Filename)
  write (6,*) 'Basename =           ', TRIM(Basename(Filename))
  write (6,*) 'Basename, w/suffix = ', TRIM(Basename(Filename,.false.))
  write (6,*) 'Dirname =            ', TRIM(Dirname(Filename))
  write (6,*) 'Reconstructed path = ', &
```

```
            TRIM(Dirname(Filename))//'/'//TRIM(Basename(Filename,.false.))

end
```

C.3 Text_Utils Module Code Listing

The main documentation of the Text_Utils Module in § 8.3 on page 60 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D409, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 10/09/07
! CVS Info:   $Id: text_utils.F90,v 1.1 2007/10/10 22:09:30 hall Exp $

module Caesar_Text_Utils_Module

  ! Global use associations.

  use Caesar_Intrinsics_Module

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Capitalize, Lowercase, Uppercase

  interface Capitalize
    module procedure Capitalize_Text_Utils
  end interface

  interface Lowercase
    module procedure Lowercase_Text_Utils
  end interface

  interface Uppercase
    module procedure Uppercase_Text_Utils
  end interface

  ! Global module parameters.

  ! Definitions for ASCII representation, others may differ.

  ! Amount to be added for capitalization.
  type(integer), parameter :: Capitalization = -32
```

```
  ! Ranges for upper- and lowercase letters.
  type(integer), dimension(2), parameter :: Majuscules = (/ 65, 90 /)
  type(integer), dimension(2), parameter :: Minuscules = (/ 97, 122 /)

contains
```

The Text_Utils Module contains the following routines which are listed in separate sections:

Capitalize_Text_Utils (§ C.3.1, page 286)

Lowercase_Text_Utils (§ C.3.2, page 287)

Uppercase_Text_Utils (§ C.3.3, page 288)

end module Caesar_Text_Utils_Module

C.3.1 Capitalize_Text_Utils Procedure

The main documentation of the Capitalize_Text_Utils Procedure in § 8.3.1 on page 60 contains additional explanation of this code listing.

```
  function Capitalize_Text_Utils (String) result(Capitalize)

    ! Input variables.

    type(character,*), intent(in) :: String         ! String to be capitalized.

    ! Output variables.

    type(character,255) :: Capitalize   ! The capitalize version of the string.

    ! Internal variables.

    type(integer) :: letter     ! Loop counter.
    type(logical) :: new_word   ! True if the next character starts a new word.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(String),5)          ! String is valid.

    ! Loop through letters, setting capitalized version.

    Capitalize = ' '
    new_word = .true.
    do letter = 1, LEN_TRIM(String)
      if (new_word) then
        Capitalize(letter:letter) = Uppercase(String(letter:letter))
      else
        Capitalize(letter:letter) = Lowercase(String(letter:letter))
      end if
      new_word = &
```

```
      (ICHAR(String(letter:letter)) .NotInInterval. Majuscules) .AND. &
      (ICHAR(String(letter:letter)) .NotInInterval. Minuscules)
  end do

  ! Verify guarantees.

  VERIFY(Valid_State(Capitalize),5)  ! Capitalize is valid.

  return
end function Capitalize_Text_Utils
```

C.3.2 Lowercase_Text_Utils Procedure

The main documentation of the Lowercase_Text_Utils Procedure in § 8.3.2 on page 60 contains additional explanation of this code listing.

```
function Lowercase_Text_Utils (String) result(Lowercase)

  ! Input variables.

  type(character,*), intent(in) :: String         ! String to be lowercased.

  ! Output variables.

  type(character,255) :: Lowercase    ! The lowercase version of the string.

  ! Internal variable.

  type(integer) :: letter             ! Loop counter.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(String),5)          ! String is valid.

  ! Loop through letters, setting lowercase version.

  Lowercase = ' '
  do letter = 1, LEN_TRIM(String)
    if (ICHAR(String(letter:letter)) .InInterval. Majuscules) then
      Lowercase(letter:letter) = &
        CHAR(ICHAR(String(letter:letter)) - Capitalization)
    else
      Lowercase(letter:letter) = String(letter:letter)
    end if
  end do

  ! Verify guarantees.

  VERIFY(Valid_State(Lowercase),5)  ! Lowercase is valid.
```

```
    return
  end function Lowercase_Text_Utils
```

C.3.3 Uppercase_Text_Utils Procedure

The main documentation of the Uppercase_Text_Utils Procedure in § 8.3.3 on page 61 contains additional explanation of this code listing.

```
  function Uppercase_Text_Utils (String) result(Uppercase)

    ! Input variables.

    type(character,*), intent(in) :: String         ! String to be uppercased.

    ! Output variables.

    type(character,255) :: Uppercase    ! The uppercase version of the string.

    ! Internal variable.

    type(integer) :: letter             ! Loop counter.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(String),5)          ! String is valid.

    ! Loop through letters, setting uppercase version.

    Uppercase = ' '
    do letter = 1, LEN_TRIM(String)
      if (ICHAR(String(letter:letter)) .InInterval. Minuscules) then
        Uppercase(letter:letter) = &
          CHAR(ICHAR(String(letter:letter)) + Capitalization)
      else
        Uppercase(letter:letter) = String(letter:letter)
      end if
    end do

    ! Verify guarantees.

    VERIFY(Valid_State(Uppercase),5)  ! Uppercase is valid.

    return
  end function Uppercase_Text_Utils
```

C.3.4 Text_Utils Module Unit Test Program

This lightly commented program performs a unit test on the Text_Utils Module, which is described in § 8.3 on page 60.

```
program Unit_Test

  use Caesar_Intrinsics_Module
  use Caesar_Text_Utils_Module
  implicit none

  type(character,80) :: String

  ! Testing statements.

  String = 'one two three four five six'
  write (6,*) ' '
  write (6,*) 'String =          ', TRIM(String)
  write (6,*) 'Lowercase =       ', TRIM(Lowercase(String))
  write (6,*) 'Capitalized =     ', TRIM(Capitalize(String))
  write (6,*) 'Uppercase =       ', TRIM(Uppercase(String))

  String = 'ONE TWO THREE FOUR FIVE SIX'
  write (6,*) ' '
  write (6,*) 'String =          ', TRIM(String)
  write (6,*) 'Lowercase =       ', TRIM(Lowercase(String))
  write (6,*) 'Capitalized =     ', TRIM(Capitalize(String))
  write (6,*) 'Uppercase =       ', TRIM(Uppercase(String))

  String = '@oNE/tWO#tHREE3fOUR:fIVE(sIX)'
  write (6,*) ' '
  write (6,*) 'String =          ', TRIM(String)
  write (6,*) 'Lowercase =       ', TRIM(Lowercase(String))
  write (6,*) 'Capitalized =     ', TRIM(Capitalize(String))
  write (6,*) 'Uppercase =       ', TRIM(Uppercase(String))

  String = 'One-Two Three-Four Five-Six'
  write (6,*) ' '
  write (6,*) 'String =          ', TRIM(String)
  write (6,*) 'Lowercase =       ', TRIM(Lowercase(String))
  write (6,*) 'Capitalized =     ', TRIM(Capitalize(String))
  write (6,*) 'Uppercase =       ', TRIM(Uppercase(String))

  String = 'A b C d E f G H I j k l m n O P Q r S T U V w X y Z'
  write (6,*) ' '
  write (6,*) 'String =          ', TRIM(String)
  write (6,*) 'Lowercase =       ', TRIM(Lowercase(String))
  write (6,*) 'Capitalized =     ', TRIM(Capitalize(String))
  write (6,*) 'Uppercase =       ', TRIM(Uppercase(String))

end
```

Appendix D

Data_Structures Module Code Listing

The main documentation of the Data_Structures Module in chapter 9 on page 63 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 08/31/99
! CVS Info:   $Id: data_structures.F90,v 4.1 2004/01/07 23:28:59 hall Exp $

module Caesar_Data_Structures_Module

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_Trace_Class
  use Caesar_Communication_Class
  use Caesar_Base_Structure_Class
  use Caesar_Data_Index_Class
  use Caesar_Assembled_Vector_Class
  use Caesar_Distributed_Vector_Class
  use Caesar_Overlapped_Vector_Class
  use Caesar_Collected_Array_Class

  ! Start up with everything untyped and public.
  ! Note: this module contains no private information.

  implicit none
  public

end module Caesar_Data_Structures_Module
```

D.1 Trace Class Code Listing

The main documentation of the Trace Class in § 9.1 on page 69 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 11/08/99
! CVS Info:   $Id: trace.F90,v 2.8 2005/03/02 01:48:19 hall Exp $

module Caesar_Trace_Class

  ! Global use associations.

  use Caesar_Intrinsics_Module
  ifdef([USE_PGSLIB],[
    use PGSLib_Module
  ])

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized

  interface Initialize
    module procedure Initialize_Trace_1
    module procedure Initialize_Trace_2
  end interface

  interface Finalize
    module procedure Finalize_Trace
  end interface

  interface Valid_State
    module procedure Valid_State_Trace
  end interface

  interface Initialized
    module procedure Initialized_Trace
  end interface

  ! Public type definitions.

  public :: Trace_type
```

```
  type Trace_type

    ! Initialization flag.

    type(logical,1) :: Initialized

    ! The number of dimensions that the index has.

    type(integer) :: Dimensionality

    ! The index values, which may be modified by the communications package.

    type(integer,1) :: Index1    ! Indirect reference indices (1-D).
    type(integer,2) :: Index2    ! Indirect reference indices (2-D).

    ! Masks for zero index values.

    type(logical,1) :: Mask1    ! Mask (1-D).
    type(logical,2) :: Mask2    ! Mask (2-D).

    ! Gather-Scatter Setup information for PGSLib.

    ifdef([USE_PGSLIB],[
      type(PGSLib_GS_Trace), pointer :: Trace
    ])

  end type Trace_type

contains
```

The Trace Class contains the following routines which are listed in separate sections:

end module Caesar_Trace_Class

D.1.1 Initialize_Trace Procedure

The main documentation of the Initialize_Trace Procedure in § 9.1.1 on page 69 contains additional explanation of this code listing.

```
  define([INITIALIZE_ROUTINE],[
    pushdef([DIM], [$1])
    pushdef([Initialize_Trace_DIM], expand(Initialize_Trace_DIM))

    subroutine Initialize_Trace_DIM (Trace, Index, Length_PE, status)
```

```
! Input variables.

type(integer,DIM,np), intent(in) :: Index ! Indirect reference indices.
type(integer) :: Length_PE ! Length of destination vector on this PE.

! Output variables.

! Trace to be initialized.
type(Trace_type), intent(out) :: Trace
type(Status_type), intent(out), optional :: status  ! Exit status.

! Internal variables.

type(Status_type), dimension(2) :: allocate_status ! Allocation Status.
type(Status_type) :: consolidated_status        ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Index),5) ! Index is valid.

! Set allocation status.

call Initialize (allocate_status)
call Initialize (consolidated_status)

! Initialize internal structures.

ifelse(DIM, [1], [
  call Initialize (Trace%Index1, SIZE(Index,1), allocate_status(1))
  call Initialize (Trace%Mask1,  SIZE(Index,1), allocate_status(2))
],[
  call Initialize (Trace%Index2, SIZE(Index,1), SIZE(Index,2), &
                   allocate_status(1))
  call Initialize (Trace%Mask2,  SIZE(Index,1), SIZE(Index,2), &
                   allocate_status(2))
])
Trace%Index[]DIM = Index
Trace%Dimensionality = DIM
Trace%Mask[]DIM = Index /= 0

! PGSLib Trace set-up.

ifdef([USE_PGSLIB],[
  NULLIFY(Trace%Trace)
  Trace%Trace => PGSLib_Setup_Trace (Trace%Index[]DIM, Length_PE, &
                                     Mask=Trace%Mask[]DIM)
])

! Process status variables.

consolidated_status = allocate_status
if (PRESENT(status)) then
```

```
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set initialization flag.

    call Initialize (Trace%Initialized, 0)

    ! Verify guarantees.

    VERIFY(Valid_State(Trace),5)  ! Trace is now valid.

    return
  end subroutine Initialize_Trace_DIM

  popdef([DIM])
  popdef([Initialize_Trace_DIM])
])

forloop([Dim],[1],[2],[
  INITIALIZE_ROUTINE(Dim)
])
```

D.1.2 Finalize_Trace Procedure

The main documentation of the Finalize_Trace Procedure in § 9.1.2 on page 72 contains additional explanation of this code listing.

```
subroutine Finalize_Trace (Trace, status)

  ! Input/Output variable.

  ! Trace to be finalized.
  type(Trace_type), intent(inout) :: Trace

  ! Output variable.

  type(Status_type), intent(out), optional :: status   ! Exit status.

  ! Internal variables.

  type(Status_type), dimension(3) :: deallocate_status ! Deallocation Status.
  type(Status_type) :: consolidated_status             ! Consolidated Status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.
```

```
    VERIFY(Valid_State(Trace),7) ! Trace is valid.

    ! Unset initialization flag.

    call Finalize (Trace%Initialized)

    ! Set deallocation status.

    call Initialize (deallocate_status)
    call Initialize (consolidated_status)

    ! PGSLib Trace deallocation.

    ifdef([USE_PGSLIB],[
      call PGSLib_Deallocate_Trace (Trace%Trace)
    ])

    ! Finalize internals.

    select case (Trace%Dimensionality)
    case (1)
      call Finalize (Trace%Index1,  deallocate_status(1))
      call Finalize (Trace%Mask1,   deallocate_status(2))
    case (2)
      call Finalize (Trace%Index2,  deallocate_status(1))
      call Finalize (Trace%Mask2,   deallocate_status(2))
    end select

    call Finalize (Trace%Dimensionality, deallocate_status(3))

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(Trace),7) ! Trace is not valid.

    return
  end subroutine Finalize_Trace
```

D.1.3 Valid_State_Trace Procedure

The main documentation of the Valid_State_Trace Procedure in § 9.1.3 on page 72 contains additional explanation of this code listing.

```
function Valid_State_Trace (Trace) result(Valid)

  ! Input variables.

  ! Variable to be checked.
  type(Trace_type), intent(in) :: Trace

  ! Output variables.

  type(logical) :: Valid  ! Logical state.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Start out true.

  Valid = .true.

  ! Check for association of pointered internals.

  Valid = Valid .and. ASSOCIATED(Trace%Initialized)
  ifdef([USE_PGSLIB],[
    Valid = Valid .and. ASSOCIATED(Trace%Trace)
  ])
  if (.not.Valid) return

  ! Check for validity of internals.

  select case (Trace%Dimensionality)
  case (1)
    Valid = Valid .and. Valid_State(Trace%Index1)
    Valid = Valid .and. Valid_State(Trace%Mask1)
  case (2)
    Valid = Valid .and. Valid_State(Trace%Index2)
    Valid = Valid .and. Valid_State(Trace%Mask2)
  end select
  if (.not.Valid) return

  ! Checks on the validity of Index.

  ! <none so far>

  return
end function Valid_State_Trace
```

D.1.4 Initialized_Trace Procedure

The main documentation of the Initialized_Trace Procedure in § 9.1.4 on page 73 contains additional explanation of this code listing.

```
  function Initialized_Trace (Trace) result(Initialized)

    ! Input variable.

    type(Trace_type), intent(in) :: Trace  ! Trace to be checked.

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = ASSOCIATED(Trace%Initialized)

    ! Verify guarantees - none.

    return
  end function Initialized_Trace
```

D.2 Communication Class Code Listing

The main documentation of the Communication Class in § 9.2 on page 73 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 11/09/98
! CVS Info:   $Id: communication.F90,v 7.14 2008/09/29 23:37:00 hall Exp $

module Caesar_Communication_Class

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_Trace_Class
  ifdef([USE_PGSLIB],[
    use pgslib_module
```

```
])

! Start up with everything untyped and private.

implicit none
private

! Public procedures.

public :: Initialize, Finalize, Valid_State
public :: Abort, Assemble, Broadcast, Distribute, Gather, Global_ALL, &
          Global_ANY, Global_Dot_Product, Global_MaxVal, Global_MinVal, &
          Global_Sum, Output, Output_Test, Parallel_Write, Scatter_AND, &
          Scatter_MAX, Scatter_MIN, Scatter_OR, Scatter_SUM

interface Initialize
  module procedure Initialize_Communication
end interface

interface Finalize
  module procedure Finalize_Communication
end interface

interface Valid_State
  module procedure Valid_State_Communication
end interface

interface Output
  module procedure Output_Communication
end interface

interface Parallel_Write
  module procedure Parallel_Write_0
  module procedure Parallel_Write_1
end interface

define([EXPLICIT_INTERFACE],[
  pushdef([PROCEDURE], [$1])
  pushdef([OP], [$2])
  pushdef([TYPE], [$3])
  pushdef([DIM], [$4])
  pushdef([PROCEDURE_OP], expand(PROCEDURE_OP))
  pushdef([PROCEDURE_OP_TYPE_DIM], expand(PROCEDURE_OP_TYPE_DIM))

  interface PROCEDURE_OP
    module procedure PROCEDURE_OP_TYPE_DIM
  end interface

  popdef([PROCEDURE])
  popdef([OP])
  popdef([TYPE])
  popdef([DIM])
  popdef([PROCEDURE_OP])
  popdef([PROCEDURE_OP_TYPE_DIM])
```

```
])

forloop([Dim],[0],[2],[
  fortext([Type],[Real Integer],[
    fortext([Op],[MaxVal MinVal Sum],[
      EXPLICIT_INTERFACE(Global, Op, Type, Dim)
    ])
  ])
  fortext([Op],[ALL ANY],[
    EXPLICIT_INTERFACE(Global, Op, Logical, Dim)
  ])
])

forloop([Dim],[1],[2],[
  fortext([Type],[Real Integer],[
    fortext([Op],[MAX MIN Sum],[
      EXPLICIT_INTERFACE(Scatter, Op, Type, Dim)
    ])
  ])
  fortext([Op],[AND OR],[
    EXPLICIT_INTERFACE(Scatter, Op, Logical, Dim)
  ])
])

fortext([Type],[Real Integer Logical],[
  interface Global_Dot_Product
    module procedure expand(Global_Dot_Product_Type)
  end interface
])

define([EXPLICIT_INTERFACE],[
  pushdef([PROCEDURE], [$1])
  pushdef([TYPE], [$2])
  pushdef([DIM], [$3])
  pushdef([PROCEDURE_TYPE_DIM], expand(PROCEDURE_TYPE_DIM))

  interface PROCEDURE
    module procedure PROCEDURE_TYPE_DIM
  end interface

  popdef([PROCEDURE])
  popdef([TYPE])
  popdef([DIM])
  popdef([PROCEDURE_TYPE_DIM])
])

forloop([Dim],[0],[3],[
  fortext([Type],[Real Integer Logical],[
    EXPLICIT_INTERFACE(Broadcast, Type, Dim)
  ])
])

forloop([Dim],[0],[2],[
  EXPLICIT_INTERFACE(Broadcast, Character, Dim)
```

```
])

forloop([Dim],[0],[1],[
  fortext([Type],[Real Integer Logical],[
    fortext([Proc],[Distribute Assemble],[
      EXPLICIT_INTERFACE(Proc, Type, Dim)
    ])
  ])
])

forloop([Dim],[0],[1],[
  fortext([Type],[Character],[
    EXPLICIT_INTERFACE(Assemble, Type, Dim)
  ])
])

forloop([Dim],[1],[2],[
  fortext([Type],[Real Integer Logical],[
    fortext([Proc],[Gather],[
      EXPLICIT_INTERFACE(Proc, Type, Dim)
    ])
  ])
])

! Public type definitions.

public :: Communication_type

type Communication_type
  ! Place holder to allow generic procedure calls -- real data
  ! associated with the communication is stored in global class
  ! variables for easy access.
  type(integer) :: i
end type Communication_type

! Public variables.

public :: delta_PE_IO_PE, IO_PE, NPEs, Parallel, Parallel_Library, &
          Serial, this_is_IO_PE, this_is_not_IO_PE, this_PE
save   :: delta_PE_IO_PE, IO_PE, NPEs, Parallel, Parallel_Library, &
          Serial, this_is_IO_PE, this_is_not_IO_PE, this_PE
ifdef([USE_PGSLIB],[
  public :: Scope
  save :: Scope
])

! Global class variables.

type(integer) :: delta_PE_IO_PE       ! Kronecker delta (PE, IO_PE).
type(integer) :: IO_PE                ! The PE number which is allowed
                                      ! to do I/O.
type(integer) :: NPEs                 ! Total number of PEs (1 for
                                      ! serial runs).
type(logical) :: Parallel             ! True for parallel runs.
```

```
  type(character,50) :: Parallel_Library ! The name of the parallel
                                         ! communication library.
  type(logical) :: Serial                ! True for serial runs.
  type(logical) :: this_is_IO_PE         ! True on the IO PE.
  type(logical) :: this_is_not_IO_PE     ! True everywhere except the IO PE.
  type(integer) :: this_PE               ! The PE number for this processor.
  ifdef([USE_PGSLIB],[
    type(PGSLib_Scope) :: Scope          ! Local or Global Scope setting.
  ])

contains
```

The Communication Class contains the following routines which are listed in separate sections:

end module Caesar_Communication_Class

D.2.1 Initialize_Communication Procedure

The main documentation of the Initialize_Communication Procedure in § 9.2.1 on page 74 contains additional explanation of this code listing.

```
  subroutine Initialize_Communication (Communication)

    ! Input/Output variable.

    ! Place holder to allow generic procedure calls -- real data associated
    ! with the communication is stored in global class variables for easy
    ! access.
    type(Communication_type), intent(inout) :: Communication
```

```
  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Initialize parallel/serial processor information.

  ifdef([USE_PGSLIB],[

    ! PGSLib Parallel Initialization.

    call PGSLib_Initialize (1)
    NPEs             = PGSLib_Inquire_nPE()
    this_PE          = PGSLib_Inquire_thisPE_Actual()
    IO_PE            = PGSLib_Inquire_IO_ROOT_PE()
    this_is_IO_PE    = PGSLib_Inquire_IO_P()
    Scope            = PGSLib_Global
    Parallel         = .true.
    Parallel_Library = 'PGSLib'

  ],[

    ! Serial initialization.

    NPEs             =  1
    this_PE          =  1
    IO_PE            =  1
    this_is_IO_PE    = .true.
    Parallel         = .false.
    Parallel_Library = 'None'

  ])

  Serial = .not.Parallel
  if (this_is_IO_PE) then
    delta_PE_IO_PE = 1
  else
    delta_PE_IO_PE = 0
  end if
  this_is_not_IO_PE = .not. this_is_IO_PE

  ! Verify guarantees.

  VERIFY(Valid_State(Communication),5)   ! Communication is now valid.

  return
end subroutine Initialize_Communication
```

D.2.2 Finalize_Communication Procedure

The main documentation of the Finalize_Communication Procedure in § 9.2.2 on page 74 contains additional explanation of this code listing.

```
  subroutine Finalize_Communication (Communication, Output_Toggle)

    ! Input variable.

    type(logical), intent(in), optional :: Output_Toggle ! Toggle final output.

    ! Input/Output variable.

    ! Place holder to allow generic procedure calls -- real data associated
    ! with the communication is stored in global class variables for easy
    ! access.
    type(Communication_type), intent(inout) :: Communication

    ! Internal variable.

    type(logical) :: IO_Output_Toggle    ! Actual output toggle.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set up output toggle.

    if (PRESENT(Output_Toggle)) then
      IO_Output_Toggle = Output_Toggle .and. this_is_IO_PE
    else
      IO_Output_Toggle = this_is_IO_PE
    end if

    ! Finalize parallel/serial processor information.

    ifdef([USE_PGSLIB],[

      ! PGSLib parallel finalization.

      if (IO_Output_Toggle) write (6,100) 'Parallel run completed.'
      call PGSLib_Finalize

    ],[

      ! Serial finalization.

      if (IO_Output_Toggle) write (6,100) 'Serial run completed.'

    ])

    ! Format statement.

100 format (/,a)

    ! Verify guarantees - none.

    return
  end subroutine Finalize_Communication
```

D.2.3 Valid_State_Communication Procedure

The main documentation of the Valid_State_Communication Procedure in § 9.2.3 on page 75 contains additional explanation of this code listing.

```
function Valid_State_Communication (Communication) result(Valid)

  ! Input variable.

  ! Place holder to allow generic procedure calls -- real data associated
  ! with the communication is stored in global class variables for easy
  ! access.
  type(Communication_type), intent(in) :: Communication

  ! Output variables.

  type(logical) :: Valid           ! Logical state.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Start out true.

  Valid = .true.

  ! Must be either serial or parallel.

  Valid = Valid .and. (Parallel .neqv. Serial)

  ! Limits on this_PE.

  Valid = Valid .and. (this_PE .InInterval. (/ 1, NPEs /) )

  ! Limits on IO_PE.

  Valid = Valid .and. (IO_PE .InInterval. (/ 1, NPEs /) )
  Valid = Valid .and. ((IO_PE == this_PE) .eqv. this_is_IO_PE)
  Valid = Valid .and. ((delta_PE_IO_PE == 1) .eqv. this_is_IO_PE)

  ! Serial-only checks.

  if (Serial) then
    Valid = Valid .and. NPEs == 1
    Valid = Valid .and. this_PE == 1
    Valid = Valid .and. IO_PE == 1
    Valid = Valid .and. this_is_IO_PE
    Valid = Valid .and. Parallel_Library == 'None'
  end if

  return
end function Valid_State_Communication
```

D.2.4 Abort Procedure

The main documentation of the Abort Procedure in § 9.2.4 on page 75 contains additional explanation of this code listing.

```
subroutine Abort ()

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Do the global abort.

  ifdef([USE_PGSLIB],[

    ! PGSLib parallel abort.

    call PGSLib_Abort ()

  ],[

    ! Serial abort.

    stop

  ])

  ! Verify guarantees - none.

  return
end subroutine Abort
```

D.2.5 Assemble Procedure

The main documentation of the Assemble Procedure in § 9.2.5 on page 75 contains additional explanation of this code listing.

```
define([ASSEMBLE_ROUTINE],[
  ifelse([$1], [character],
    [pushdef([TYPE], [character,*])],
    [pushdef([TYPE], [$1])])
  pushdef([DIM], [$2])
  pushdef([Assemble_TYPE_DIM], expand(Assemble_$1_DIM))

  subroutine Assemble_TYPE_DIM (Output, Input)

    ! Input variables.

    type(TYPE,DIM,np), intent(in) :: Input    ! Variable to be assembled.

    ! Output variable.
```

```
    type(TYPE,1,np), intent(out) :: Output    ! Assembled variable.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Do the global assemble.

    ifdef([USE_PGSLIB],[

      ! PGSLib parallel assemble.

      call PGSLib_Collate (Output, Input)

    ],[

      ! Serial assemble.

      Output = Input

    ])

    ! Verify guarantees - none.

    return
  end subroutine Assemble_TYPE_DIM

  popdef([TYPE])
  popdef([DIM])
  popdef([Assemble_TYPE_DIM])
])

forloop([Dim],[0],[1],[
  fortext([Type],[real integer logical character],[
    ASSEMBLE_ROUTINE(Type, Dim)
  ])
])
```

D.2.6 Broadcast Procedure

The main documentation of the Broadcast Procedure in § 9.2.6 on page 76 contains additional explanation of this code listing.

```
define([BROADCAST_ROUTINE],[
  ifelse([$1], [character],
    [pushdef([TYPE], [character,*])],
    [pushdef([TYPE], [$1])])
  pushdef([DIM], [$2])
  pushdef([Broadcast_TYPE_DIM], expand(Broadcast_$1_DIM))

  subroutine Broadcast_TYPE_DIM (Variable)
```

```
    ! Input/Output variable.

    type(TYPE,DIM,np), intent(inout) :: Variable  ! Variable to be broadcast.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Do the global broadcast.

    ifdef([USE_PGSLIB],[

      ! PGSLib parallel broadcast.

      call PGSLib_BCast (Variable)

    ],[

      ! Serial broadcast - no op.

    ])

    ! Verify guarantees - none.

    return
  end subroutine Broadcast_TYPE_DIM

  popdef([TYPE])
  popdef([DIM])
  popdef([Broadcast_TYPE_DIM])
])

forloop([Dim],[0],[3],[
  fortext([Type],[real integer logical],[
    BROADCAST_ROUTINE(Type, Dim)
  ])
])

forloop([Dim],[0],[2],[
  BROADCAST_ROUTINE(character, Dim)
])
```

D.2.7 Distribute Procedure

The main documentation of the Distribute Procedure in § 9.2.7 on page 76 contains additional explanation of this code listing.

```
define([DISTRIBUTE_ROUTINE],[
  pushdef([TYPE], [$1])
  pushdef([Distribute_TYPE_0], expand(Distribute_TYPE_0))
```

```
  subroutine Distribute_TYPE_0 (Output, Input)

    ! Input variables.

    type(TYPE,1,np), intent(in) :: Input    ! Variable to be distributed.

    ! Output variable.

    type(TYPE), intent(out) :: Output       ! Distributed variable.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    ! Input is the right size.
    VERIFY(SIZE(Input) == NPEs * delta_PE_IO_PE,3)

    ! Do the global distribute.

    ifdef([USE_PGSLIB],[

      ! PGSLib parallel distribute.

      call PGSLib_Dist (Output, Input)

    ],[

      ! Serial distribute.

      Output = Input(1)

    ])

    ! Verify guarantees - none.

    return
  end subroutine Distribute_TYPE_0

  popdef([TYPE])
  popdef([Distribute_TYPE_0])
])

fortext([Type],[real integer logical],[
  DISTRIBUTE_ROUTINE(Type)
])

define([DISTRIBUTE_ROUTINE],[
  pushdef([TYPE], [$1])
  pushdef([Distribute_TYPE_1], expand(Distribute_TYPE_1))

  subroutine Distribute_TYPE_1 (Output, Input, Lengths)

    ! Input variables.
```

```
    type(integer,1,np), intent(in) :: Lengths ! Distribution lengths.
    type(TYPE,1,np), intent(in) :: Input      ! Variable to be distributed.

    ! Output variable.

    type(TYPE,1,np), intent(out) :: Output    ! Distributed variable.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    ! Input is the right size.
    VERIFY(SIZE(Input) == SUM(Lengths) * delta_PE_IO_PE,3)
    ! Lengths is the right size.
    VERIFY(SIZE(Lengths) == NPEs,3)

    ! Do the global distribute.

    ifdef([USE_PGSLIB],[

      ! PGSLib parallel distribute.

      call PGSLib_Dist (Output, Input, Lengths)

    ],[

      ! Serial distribute.

      Output = Input

    ])

    ! Verify guarantees - none.

    return
  end subroutine Distribute_TYPE_1

  popdef([TYPE])
  popdef([Distribute_TYPE_1])
])

fortext([Type],[real integer logical],[
  DISTRIBUTE_ROUTINE(Type)
])
```

D.2.8 Gather Procedure

The main documentation of the Gather Procedure in § 9.2.8 on page 76 contains additional explanation of this code listing.

```
define([GATHER_ROUTINE],[
  pushdef([TYPE], [$1])
```

```
pushdef([DIM], [$2])
pushdef([Gather_TYPE_DIM], expand(Gather_TYPE_DIM))

subroutine Gather_TYPE_DIM (Output, Input, Index, Trace)

  ! Input variables.

  ! Distributed vector (bare naked vector) to be gathered.
  type(TYPE,1,np), intent(in) :: Input
  type(integer,DIM,np), intent(in), optional :: Index ! Indirect reference
                                                      ! indices.

  ! Input/Output variable.

  type(Trace_type), intent(inout), optional :: Trace  ! Setup information.

  ! Output variable.

  type(TYPE,DIM,np), intent(out) :: Output            ! Gathered variable.

  ! Internal variable.

  ifdef([USE_PGSLIB],[
    type(integer,DIM) :: Index_tmp           ! Index temporary.
    type(logical,DIM) :: Mask_tmp            ! Index mask temporary.
  ],[
    ifelse(DIM, [1], [], [
      type(integer) :: column                ! Loop parameter.
    ])
    type(integer) :: row                     ! Loop parameter.
  ])

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  ! Either the Trace or the Index, or both, must be present.
  VERIFY(PRESENT(Index) .or. PRESENT(Trace),5)

  ! Initialize the Trace.

  if (PRESENT(Trace)) then
    if (.not. Initialized(Trace)) then
      VERIFY(PRESENT(Index),5)
      call Initialize (Trace, Index, SIZE(Input))
    end if
  end if

  ! Do the global gather.

  ifdef([USE_PGSLIB],[

    ! PGSLib parallel gather.
```

```
    if (PRESENT(Trace)) then
      call PGSLib_Gather (Output, Input, Trace%Index[]DIM,  &
                          Trace%Trace, Trace%Mask[]DIM)
    else
      ifelse(DIM, [1], [
        call Initialize (Index_tmp, SIZE(Index))
        call Initialize (Mask_tmp,  SIZE(Index))
      ], [
        call Initialize (Index_tmp, SIZE(Index,1), SIZE(Index,2))
        call Initialize (Mask_tmp,  SIZE(Index,1), SIZE(Index,2))
      ])
      Index_tmp = Index
      Mask_tmp = Index_tmp /= 0
      call PGSLib_Gather (Output, Input, Index_tmp, Mask=Mask_tmp)
      call Finalize (Index_tmp)
      call Finalize (Mask_tmp)
    end if

  ],[

    ! Serial gather.

    if (PRESENT(Trace)) then
      ifelse(DIM, [1], [
        do row = 1, SIZE(Output, 1)
          if (Trace%Index1(row) /= 0) then
            Output(row) = Input(Trace%Index1(row))
          end if
        end do
      ],[
        do column = 1, SIZE(Output, 2)
          do row = 1, SIZE(Output, 1)
            if (Trace%Index2(row,column) /= 0) then
              Output(row,column) = Input(Trace%Index2(row,column))
            end if
          end do
        end do
      ])
    else
      ifelse(DIM, [1], [
        do row = 1, SIZE(Output, 1)
          if (Index(row) /= 0) then
            Output(row) = Input(Index(row))
          end if
        end do
      ],[
        do column = 1, SIZE(Output, 2)
          do row = 1, SIZE(Output, 1)
            if (Index(row,column) /= 0) then
              Output(row,column) = Input(Index(row,column))
            end if
          end do
        end do
      ])
```

```
      end if
    ])

    ! Verify guarantees - none.

    return
  end subroutine Gather_TYPE_DIM

  popdef([TYPE])
  popdef([DIM])
  popdef([Gather_TYPE_DIM])
])

forloop([Dim],[1],[2],[
  fortext([Type],[real integer logical],[
    GATHER_ROUTINE(Type, Dim)
  ])
])
```

D.2.9 Global Reduction Functions

The main documentation of the Global Reduction Functions in § 9.2.9 on page 77 contains additional explanation of this code listing.

```
define([REDUCTION_ROUTINE],[
  pushdef([REDUCTION], [$1])
  pushdef([TYPE], [$2])
  pushdef([DIM], [$3])
  pushdef([Global_REDUCTION_TYPE_DIM], expand(Global_REDUCTION_TYPE_DIM))
  pushdef([PGSLib_Global_REDUCTION], expand(PGSLib_Global_REDUCTION))

  function Global_REDUCTION_TYPE_DIM (Input) result(Output)

    ! Input variable.

    type(TYPE,DIM,np), intent(in) :: Input  ! Variable to be reduced.

    ! Output variable.

    type(TYPE) :: Output  ! Reduction result.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Do the global reduction.

    ifdef([USE_PGSLIB],[

      ! PGSLib parallel reduction.

      Output = PGSLib_Global_REDUCTION (Input)
```

```
    ],[

      ! Serial reduction.

      Output = REDUCTION (Input)

    ])

    ! Verify guarantees - none.

    return
  end function Global_REDUCTION_TYPE_DIM

  popdef([TYPE])
  popdef([REDUCTION])
  popdef([DIM])
  popdef([Global_REDUCTION_TYPE_DIM])
  popdef([PGSLib_Global_REDUCTION])
])

forloop([Dim],[0],[2],[
  fortext([Type],[real integer],[
    fortext([Op],[Sum MaxVal MinVal],[
      REDUCTION_ROUTINE(Op, Type, Dim)
    ])
  ])
  fortext([Op],[ALL ANY],[
    REDUCTION_ROUTINE(Op, logical, Dim)
  ])
])

define([DOT_PRODUCT_ROUTINE],[
  pushdef([TYPE], [$1])
  pushdef([Global_Dot_Product_TYPE], expand(Global_Dot_Product_TYPE))

  function Global_Dot_Product_TYPE (Input1, Input2) result(Output)

    ! Input variable.

    type(TYPE,1,np), intent(in) :: Input1 ! Variable to be reduced.
    type(TYPE,1,np), intent(in) :: Input2 ! Variable to be reduced.

    ! Output variable.

    type(TYPE) :: Output  ! Dot_Product result.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Do the global dot_product.

    ifdef([USE_PGSLIB],[
```

```
      ! PGSLib parallel dot_product.

      Output = PGSLib_Global_Dot_Product (Input1, Input2)

    ],[

      ! Serial dot_product.

      Output = Dot_Product (Input1, Input2)

    ])

    ! Verify guarantees - none.

    return
  end function Global_Dot_Product_TYPE

  popdef([TYPE])
  popdef([Global_Dot_Product_TYPE])
])

fortext([Type],[real integer logical],[
  DOT_PRODUCT_ROUTINE(Type)
])
```

D.2.10 Output_Communication Procedure

The main documentation of the Output_Communication Procedure in § 9.2.10 on page 78 contains additional explanation of this code listing.

```
subroutine Output_Communication (Communication, Unit)

  ! Input variables.

  ! Place holder to allow generic procedure calls -- real data associated
  ! with the communication is stored in global class variables for easy
  ! access.
  type(Communication_type), intent(in) :: Communication
  type(integer), intent(in), optional :: Unit          ! Output unit.

  ! Internal variable.

  type(integer) :: A_Unit                              ! Actual output unit.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Communication),5)      ! Communication is valid.

  ! Only output on the IO PE.
```

```
    if (this_is_IO_PE) then

      ! Set unit number.

      if (PRESENT(Unit)) then
        A_Unit = Unit
      else
        A_Unit = 6
      end if

      ! Output Identification Info.

      if (Parallel) then
        write (A_Unit,100) 'Running in parallel mode using ', &
                           TRIM(Parallel_Library), ':'
        write (A_Unit,101) 'NPEs  = ', NPEs
        write (A_Unit,101) 'PE    = ', this_PE
        write (A_Unit,101) 'IO_PE = ', IO_PE
      else
        write (A_Unit,100) 'Running in serial mode.'
      end if

    end if

    ! Format statements.

100 format (a,:,a,a,/)
101 format (2x,a,i5)

    ! Verify guarantees - none.

    return
  end subroutine Output_Communication
```

D.2.11 Output_Test Procedure

The main documentation of the Output_Test Procedure in § 9.2.11 on page 78 contains additional explanation of this code listing.

```
  subroutine Output_Test (Test_Name, Success, Unit)

    ! Input variables.

    type(integer), intent(in), optional :: Unit    ! Output unit.
    type(logical) :: Success                       ! The result of the test.
    type(character,*) :: Test_Name                 ! The name of the test.

    ! Internal variable.

    type(integer) :: A_Unit                        ! Actual output unit.
```

```
    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Only output on the IO PE.

    if (this_is_IO_PE) then

      ! Set unit number.

      if (PRESENT(Unit)) then
        A_Unit = Unit
      else
        A_Unit = 6
      end if

      ! Output test result.

      if (Success) then
        write (A_Unit,100) Test_Name, ' check successful.'
      else
        write (A_Unit,100) Test_Name, ' check failed.'
      end if

    end if

    ! Format statement.

100 format (/, 2x, 2a)

    ! Verify guarantees - none.

    return
  end subroutine Output_Test
```

D.2.12 Parallel_Write Procedure

The main documentation of the Parallel_Write Procedure in § 9.2.12 on page 78 contains additional explanation of this code listing.

```
  subroutine Parallel_Write_0 (String, Unit, PE)

    ! Input variables.

    type(character,*), intent(in) :: String      ! Output string.
    type(integer), intent(in), optional :: Unit  ! Output unit number.
    type(integer), intent(in), optional :: PE    ! Output PE number.

    ! Internal variables.

    type(character,LEN(String),1) :: IO_String  ! Assembled output string.
    type(integer) :: IO_Unit                    ! Actual output unit number.
```

```
    type(integer) :: i                          ! Loop counter.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Initialize (on all PEs) and assemble the IO PE variable.

    call Initialize (IO_String, NPEs)
    call Assemble (IO_String, String)

    ! Only output on the IO PE.

    if (this_is_IO_PE) then

      ! Set unit number.

      if (PRESENT(Unit)) then
        IO_Unit = Unit
      else
        IO_Unit = 6
      end if

      ! Output for a single PE.

      if (PRESENT(PE)) then
        write (IO_Unit,100) TRIM(IO_String(PE))

      ! Output for all the PEs.

      else
        do i = 1, NPEs
          write (IO_Unit,100) TRIM(IO_String(i))
        end do
      end if

      ! Format statement.

100   format (a)

    end if

    ! Finalize variable.

    call Finalize (IO_String)

    ! Verify guarantees - none.

    return
  end subroutine Parallel_Write_0

  subroutine Parallel_Write_1 (String, Unit, PE)

    ! Input variables.
```

```
    type(character,*,1,np), intent(in) :: String ! Output string.
    type(integer), intent(in), optional :: Unit  ! Output unit number.
    type(integer), intent(in), optional :: PE    ! Output PE number.

    ! Internal variables.

    type(character,LEN(String),1) :: IO_String ! Assembled output string.
    type(integer) :: IO_Unit                   ! Actual output unit number.
    type(integer) :: i                         ! Loop counter.
    type(integer,1) :: Length_Vector         ! Vector of lengths for each PE.
    type(integer) :: Length_PE                 ! Lengths on each PE.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Initialize (on all PEs) and assemble the IO PE variables.

    Length_PE = SIZE(String)
    call Initialize (Length_Vector, NPEs)
    call Assemble (Length_Vector, Length_PE)
    call Initialize (IO_String, SUM(Length_Vector))
    call Assemble (IO_String, String)

    ! Only output on the IO PE.

    if (this_is_IO_PE) then

      ! Set unit number.

      if (PRESENT(Unit)) then
        IO_Unit = Unit
      else
        IO_Unit = 6
      end if

      ! Output for a single PE.

      if (PRESENT(PE)) then
        do i = SUM(Length_Vector(1:PE-1)) + 1, SUM(Length_Vector(1:PE))
          write (IO_Unit,100) TRIM(IO_String(i))
        end do

      ! Output for all the PEs.

      else
        do i = 1, SUM(Length_Vector)
          write (IO_Unit,100) TRIM(IO_String(i))
        end do
      end if

      ! Format statement.
```

```
100   format (a)

    end if

    ! Finalize variables.

    call Finalize (Length_Vector)
    call Finalize (IO_String)

    ! Verify guarantees - none.

    return
  end subroutine Parallel_Write_1
```

D.2.13 Scatter Procedure

The main documentation of the Scatter Procedure in § 9.2.13 on page 79 contains additional explanation of this code listing.

```
  define([SCATTER_ROUTINE],[
    pushdef([TYPE], [$1])
    pushdef([DIM], [$2])
    pushdef([OP], [$3])
    pushdef([Scatter_OP_TYPE_DIM], expand(Scatter_OP_TYPE_DIM))
    pushdef([PGSLib_Scatter_OP], expand(PGSLib_Scatter_OP))

    subroutine Scatter_OP_TYPE_DIM (Output, Input, Index, Trace)

      ! Input variables.

      ! Distributed vector (bare naked vector) to be scattered.
      type(TYPE,DIM,np), intent(in) :: Input
      type(integer,DIM,np), intent(in), optional :: Index ! Indirect reference
                                                          ! indices.

      ! Input/Output variable.

      type(Trace_type), intent(inout), optional :: Trace   ! Setup information.

      ! Output variable.

      type(TYPE,1,np), intent(out) :: Output     ! Scattered variable.

      ! Internal variables.

      ifdef([USE_PGSLIB],[
        type(integer,DIM) :: Index_tmp           ! Index temporary.
        type(logical,DIM) :: Mask_tmp            ! Index mask temporary.
      ],[
        ifelse(DIM, [1], [], [
          type(integer) :: column                ! Loop parameter.
        ])
```

```
  type(integer) :: row                    ! Loop parameter.
])

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

! Either the Trace or the Index, or both, must be present.
VERIFY(PRESENT(Index) .or. PRESENT(Trace),1)

! Initialize the Trace.

if (PRESENT(Trace)) then
  if (.not. Initialized(Trace)) then
    VERIFY(PRESENT(Index),5)
    call Initialize (Trace, Index, SIZE(Input))
  end if
end if

! Do the global scatter.

ifdef([USE_PGSLIB],[

  ! PGSLib parallel scatter.

  if (PRESENT(Trace)) then
    call PGSLib_Scatter_OP (Output, Input, Trace%Index[]DIM, &
                            Trace%Trace, Trace%Mask[]DIM)
  else
    ifelse(DIM, [1], [
      call Initialize (Index_tmp, SIZE(Index))
      call Initialize (Mask_tmp,  SIZE(Index))
    ], [
      call Initialize (Index_tmp, SIZE(Index,1), SIZE(Index,2))
      call Initialize (Mask_tmp,  SIZE(Index,1), SIZE(Index,2))
    ])
    Index_tmp = Index
    Mask_tmp = Index_tmp /= 0
    call PGSLib_Scatter_OP (Output, Input, Index_tmp, Mask=Mask_tmp)
    call Finalize (Index_tmp)
    call Finalize (Mask_tmp)
  end if

],[

  ! Serial scatter.

  if (PRESENT(Trace)) then
    ifelse(DIM, [1], [
      do row = 1, SIZE(Input, 1)
        if (Trace%Index1(row) /= 0) then
          Output(Trace%Index1(row)) = &
            OPERATION(Output(Trace%Index1(row)), Input(row))
        end if
```

```
          end do
        ],[
          do column = 1, SIZE(Input, 2)
            do row = 1, SIZE(Input, 1)
              if (Trace%Index2(row,column) /= 0) then
                Output(Trace%Index2(row,column)) = &
                  OPERATION(Output(Trace%Index2(row,column)), &
                            Input(row,column))
              end if
            end do
          end do
        ])
      else
        ifelse(DIM, [1], [
          do row = 1, SIZE(Input, 1)
            if (Index(row) /= 0) then
              Output(Index(row)) = OPERATION(Output(Index(row)), Input(row))
            end if
          end do
        ],[
          do column = 1, SIZE(Input, 2)
            do row = 1, SIZE(Input, 1)
              if (Index(row,column) /= 0) then
                Output(Index(row,column)) = &
                  OPERATION(Output(Index(row,column)), Input(row,column))
              end if
            end do
          end do
        ])
      end if

    ])

    ! Verify guarantees - none.

    return
  end subroutine Scatter_OP_TYPE_DIM

  popdef([TYPE])
  popdef([DIM])
  popdef([OP])
  popdef([Scatter_OP_TYPE_DIM])
  popdef([PGSLib_Scatter_OP])
])

forloop([Dim],[1],[2],[
  fortext([Op],[SUM MAX MIN],[
    ifelse(Op, [SUM], [
      pushdef([OPERATION], [$1 + $2])
    ], [
      pushdef([OPERATION], [Op[]($1, $2)])
    ])
    fortext([Type],[real integer],[
      SCATTER_ROUTINE(Type, Dim, Op)
```

```
      ])
    ])
    fortext([Op],[AND OR],[
      pushdef([OPERATION], [$1 .Op. $2])
      SCATTER_ROUTINE(logical, Dim, Op)
    ])
  ])
  popdef([OPERATION])
```

D.2.14 Communication Class Unit Test Program

This lightly commented program performs a unit test on the Communication Class, which is described in § 9.2 on page 73.

```
program Unit_Test

  use Caesar_Intrinsics_Module
  use Caesar_Communication_Class
  implicit none

  type(real,1) :: R, R2
  type(real,1) :: R_PE
  type(integer,1) :: Length_PE
  type(integer) :: already_counted, i, index, Length, pe
  type(Communication_type) :: Comm

  ! Initialize communications.

  call Initialize (Comm)
  call Output (Comm)

  ! Set total length and individual lengths of parallel vectors.

  Length = 100
  call Initialize (Length_PE, NPEs)
  ! Unequal setting per PE to trip more possible bugs.
  Length_PE(NPEs) = Length
  do pe = 1, NPEs-1
    Length_PE(pe) = (Length / NPEs) / 2
    Length_PE(NPEs) = Length_PE(NPEs) - Length_PE(pe)
  end do

  ! Initialize assembled and distributed vectors.

  if (this_is_IO_PE) then
    call Initialize (R, Length)
    call Initialize (R2, Length)
  else
    call Initialize (R, 0)
    call Initialize (R2, 0)
  end if
  call Initialize (R_PE, Length_PE(this_PE))
```

```
  ! Set the assembled vector.

  if (this_is_IO_PE) then
    R = (/ ( changetype(real, Length - i), i=1,Length ) /)
  end if

  ! Distribute the vector.

  call Distribute (R_PE, R, Length_PE)

  ! Check the distributed vector.

  if (NPEs == 1) then
    already_counted = 0
  else
    already_counted = SUM( Length_PE(1:this_PE-1) )
  end if
  do i = 1, Length_PE(this_PE)
    index = already_counted + i
    if (R_PE(i) /= changetype(real, Length - index) ) then
      write (6,*) 'Error --> ', R_PE(i), R(i), i, Length - index
    end if
  end do

  ! Assemble and check the vector.

  call Assemble (R2, R_PE)
  if (ANY(R2 /= R) .and. this_is_IO_PE) then
    write (6,*) 'Error --> R /= R2'
  end if

  ! Check state of communication.

  VERIFY(Valid_State(Comm),0)

  ! Finalize communications.

  call Finalize (Comm)

end
```

D.3 Base_Structure Class Code Listing

The main documentation of the Base_Structure Class in § 9.3 on page 79 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
```

```
!         email: Hall@LANL.gov
!
! Created on: 09/13/99
! CVS Info:   $Id: base_structure.F90,v 5.18 2009/08/26 00:40:00 hall Exp $

module Caesar_Base_Structure_Class

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_Communication_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Generate_Even_Distribution, Generate_Multiple, First_PE, Last_PE, &
            Length_PE, Length_Total, Length_Vector, Locus, Output, Range_PE

  interface Initialize
    module procedure Initialize_Base_Structure
  end interface

  interface Finalize
    module procedure Finalize_Base_Structure
  end interface

  interface Valid_State
    module procedure Valid_State_Base_Structure
  end interface

  interface Initialized
    module procedure Initialized_Base_Structure
  end interface

  interface Generate_Even_Distribution
    module procedure Generate_Even_Distribution
  end interface

  interface Generate_Multiple
    module procedure Generate_Multiple_Base_Struct
  end interface

  interface Output
    module procedure Output_Base_Structure
  end interface

  fortext([Value],[First_PE Last_PE Length_PE Length_Total dnl
                   Length_Vector Locus Range_PE],[
    interface Value
```

```
      module procedure expand(Get_Value_Structure)
    end interface
  ])

  ! Public variables.

  public :: name_length

  type(integer), parameter :: &
    name_length = 72   ! Length of the character strings for names.

  ! Public type definitions.

  public :: Base_Structure_type

  type Base_Structure_type

    ! Initialization flag.

    type(integer) :: Initialized

    ! The location or variable name which is distributed over the processors
    ! (Cells, Nodes, Faces, Equations, Variables, etc.).

    type(character,name_length) :: Locus

    ! Total length of the distributed axis of the entire vector (including
    ! all PEs).

    type(integer) :: Length_Total

    ! Length of the distributed axis on this PE.

    type(integer) :: Length_PE

    ! A vector containing the length of the distributed axis for each PE.

    type(integer,1) :: Length_Vector

    ! First, last and range of global index numbers for this PE.

    type(integer) :: First_PE
    type(integer) :: Last_PE
    type(integer), dimension(2) :: Range_PE

  end type Base_Structure_type


contains
```

The Base_Structure Class contains the following routines which are listed in separate sections:

Initialize_Base_Structure (§ D.3.1, page 327)

end module Caesar_Base_Structure_Class

D.3.1 Initialize_Base_Structure Procedure

The main documentation of the Initialize_Base_Structure Procedure in § 9.3.1 on page 80 contains additional explanation of this code listing.

```
subroutine Initialize_Base_Structure (Structure, Length_Vector, Locus, &
                                      status)

  ! Use associations.

  use Caesar_Flags_Module, only: initialized_flag

  ! Input variables.

  type(integer,1) :: Length_Vector                 ! Length for each PE.
  type(character,*), intent(in), optional :: Locus ! Distributed location.

  ! Output variables.

  ! Base_Structure to be initialized.
  type(Base_Structure_type), intent(out) :: Structure
  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(Status_type) :: allocate_status ! Allocation Status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(SIZE(Length_Vector)==NPEs,5)   ! Length_Vector is the right size.
  VERIFY(Valid_State(Length_Vector),5)  ! Length_Vector is valid.

  ! Allocations and initializations.

  call Initialize (allocate_status)
  call Initialize (Structure%Length_Vector, NPEs, allocate_status)
  if (PRESENT(status)) then
```

```
      WARN_IF(Error(allocate_status), 5)
      status = allocate_status
    else
      VERIFY(Normal(allocate_status), 5)
    end if
    call Finalize (allocate_status)

    ! Set up internals.

    if (PRESENT(Locus)) Structure%Locus = Locus
    Structure%Length_Vector = Length_Vector
    Structure%Length_Total = SUM(Length_Vector)
    Structure%Length_PE = Length_Vector(this_PE)
    Structure%Last_PE = SUM(Length_Vector(1:this_PE))
    Structure%First_PE = Structure%Last_PE - Structure%Length_PE + 1
    Structure%Range_PE = (/ Structure%First_PE, Structure%Last_PE /)

    ! Set initialization flag.

    Structure%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(Structure),5)  ! Structure is now valid.

    return
  end subroutine Initialize_Base_Structure
```

D.3.2 Finalize_Base_Structure Procedure

The main documentation of the Finalize_Base_Structure Procedure in § 9.3.2 on page 81 contains additional explanation of this code listing.

```
  subroutine Finalize_Base_Structure (Structure, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Base_Structure to be finalized.
    type(Base_Structure_type), intent(inout) :: Structure

    ! Output variable.

    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(8) :: deallocate_status ! Deallocation Status.
    type(Status_type) :: consolidated_status             ! Consolidated Status.
```

```
  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Structure),7) ! Structure is valid.

  ! Unset initialization flag.

  Structure%Initialized = uninitialized_flag

  ! Set deallocation status.

  call Initialize (deallocate_status)
  call Initialize (consolidated_status)

  ! Finalize internals.

  call Finalize (Structure%Length_Vector, deallocate_status(1))
  call Finalize (Structure%Locus,         deallocate_status(2))
  call Finalize (Structure%Length_Total,  deallocate_status(3))
  call Finalize (Structure%Length_PE,     deallocate_status(4))
  call Finalize (Structure%Last_PE,       deallocate_status(5))
  call Finalize (Structure%First_PE,      deallocate_status(6))
  call Finalize (Structure%Range_PE(1),   deallocate_status(7))
  call Finalize (Structure%Range_PE(2),   deallocate_status(8))

  ! Process status variables.

  consolidated_status = deallocate_status
  if (PRESENT(status)) then
    WARN_IF(Error(consolidated_status), 5)
    status = consolidated_status
  else
    VERIFY(Normal(consolidated_status), 5)
  end if
  call Finalize (consolidated_status)
  call Finalize (deallocate_status)

  ! Verify guarantees.

  VERIFY(.not.Valid_State(Structure),7) ! Structure is not valid.

  return
end subroutine Finalize_Base_Structure
```

D.3.3 Valid_State_Base_Structure Procedure

The main documentation of the Valid_State_Base_Structure Procedure in § 9.3.3 on page 81 contains additional explanation of this code listing.

```
function Valid_State_Base_Structure (Structure) result(Valid)
```

```
    ! Input variables.

    ! Variable to be checked.
    type(Base_Structure_type), intent(in) :: Structure

    ! Output variables.

    type(logical) :: Valid       ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for validity of internals.

    Valid = Valid .and. Initialized(Structure)
    Valid = Valid .and. Valid_State(Structure%Length_Vector)
    Valid = Valid .and. Valid_State(Structure%Locus)
    Valid = Valid .and. Valid_State(Structure%Length_Total)
    Valid = Valid .and. Valid_State(Structure%Length_PE)
    Valid = Valid .and. Valid_State(Structure%Last_PE)
    Valid = Valid .and. Valid_State(Structure%First_PE)
    if (.not.Valid) return

    ! Checks on the validity of Structure.

    Valid = Valid .and. &
            Structure%Length_Total == SUM(Structure%Length_Vector)
    Valid = Valid .and. &
            Structure%Length_PE == Structure%Length_Vector(this_PE)
    Valid = Valid .and. &
            Structure%Last_PE == SUM(Structure%Length_Vector(1:this_PE))
    Valid = Valid .and. &
            Structure%First_PE == Structure%Last_PE - Structure%Length_PE + 1
    Valid = Valid .and. Structure%Range_PE(1) == Structure%First_PE
    Valid = Valid .and. Structure%Range_PE(2) == Structure%Last_PE

    return
  end function Valid_State_Base_Structure
```

D.3.4 Initialized_Base_Structure Procedure

The main documentation of the Initialized_Base_Structure Procedure in § 9.3.4 on page 82 contains additional explanation of this code listing.

```
  function Initialized_Base_Structure (Structure) result(Initialized)

    ! Use associations.
```

```
    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! Base_Structure to be checked.
    type(Base_Structure_type), intent(in) :: Structure

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = Structure%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Base_Structure
```

D.3.5 Generate_Even_Distribution Procedure

The main documentation of the Generate_Even_Distribution Procedure in § 9.3.5 on page 82 contains additional explanation of this code listing.

```
  subroutine Generate_Even_Distribution (Vector, NItems)

    ! Input variable.

    ! Number of items to be evenly distributed.
    type(integer), intent(in) :: NItems

    ! Input/Output variable.

    ! Vector to hold the even distribution.
    type(integer,1) :: Vector

    ! Internal variables.

    type(integer) :: extra   ! Leveling variable.
    type(integer) :: i       ! Loop counter.
    type(integer) :: NSlots  ! Number of slots available in the Vector.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Vector),5)  ! Vector is valid.
```

```
    VERIFY(Valid_State(NItems),5)  ! NItems is valid.
    VERIFY(NItems>=SIZE(Vector),5) ! NItems is big enough.

    ! Generate the even distribution.

    NSlots = SIZE(Vector)
    do i = 1, NSlots-1
      Vector(i) = NItems / NSlots
    end do
    Vector(NSlots) = NItems - SUM(Vector(1:NSlots-1))
    extra = Vector(NSlots) - NItems / NSlots
    do i = 1, extra
      Vector(i) = Vector(i) + 1
    end do
    Vector(NSlots) = NItems - SUM(Vector(1:NSlots-1))

    ! Verify guarantees.

    VERIFY(Valid_State(Vector),5)              ! Vector is valid.
    VERIFY(SUM(Vector)==NItems,5)              ! Vector sum is correct.
    VERIFY(MAXVAL(Vector)-MINVAL(Vector)<=1,5) ! Vector is evenly distributed.

    return
  end subroutine Generate_Even_Distribution
```

D.3.6 Generate_Multiple_Base_Structure Procedure

The main documentation of the Generate_Multiple_Base_Structure Procedure in § 9.3.6 on page 82 contains additional explanation of this code listing.

```
  subroutine Generate_Multiple_Base_Struct (Structure_Multiple, N, &
                                            Structure_source, Locus, status)

    ! Input variables.

    type(Base_Structure_type), intent(in) :: Structure_source
    type(integer), intent(in) :: N
    type(character,*), intent(in), optional :: Locus ! Distributed location.

    ! Output variables.

    ! Base_Structure to be initialized.
    type(Base_Structure_type), intent(out) :: Structure_Multiple
    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(2) :: allocate_status ! Allocation Status.
    type(Status_type) :: consolidated_status           ! Consolidated Status.
    type(integer,1) :: Length_Vector                   ! Length for each PE.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ! Verify requirements.

    VERIFY(Valid_State(Structure_source),5)        ! Structure_source is valid.

    ! Allocations and initializations.

    call Initialize (allocate_status)
    call Initialize (consolidated_status)
    call Initialize (Length_Vector, NPEs, allocate_status(1))

    ! Multiply the length vector.

    Length_Vector = N * Structure_source%Length_Vector

    ! Initialize Structure_Multiple to be a multiple of Structure_source.

    call Initialize (Structure_Multiple, Length_Vector, &
                     status=allocate_status(2))

    ! Set Locus.

    if (PRESENT(Locus)) then
      Structure_Multiple%Locus = Locus
    else
      Structure_Multiple%Locus = Structure_source%Locus
    end if

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Finalize variable.

    call Finalize (Length_Vector)

    ! Verify guarantees.

    VERIFY(Valid_State(Structure_Multiple),5) ! Base_Structure is now valid.

    return
  end subroutine Generate_Multiple_Base_Struct
```

D.3.7 Get Value Base_Structure Functions

The main documentation of the Get Value Base_Structure Functions in § 9.3.7 on page 83 contains additional explanation of this code listing.

```
define([ACCESS_ROUTINE],[
  pushdef([VALUE], [$1])
  pushdef([Get_VALUE_Structure], expand(Get_VALUE_Structure))

  function Get_VALUE_Structure (Structure) result(VALUE)

    ! Input/Output variables.

    ! Base_Structure object.
    type(Base_Structure_type), intent(in) :: Structure

    ! Output variables.

    type(integer) :: VALUE               ! Base_Structure value to be output.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Structure),5)     ! Base_Structure is valid.

    ! Set value.

    VALUE = Structure%VALUE

    ! Verify guarantees - none.

    return
  end function Get_VALUE_Structure

  popdef([VALUE])
  popdef([Get_VALUE_Structure])
])

fortext([Value],
        [First_PE Last_PE Length_PE Length_Total],[
  ACCESS_ROUTINE(Value)
])

function Get_Length_Vector_Structure (Structure) result(Length_Vector)

  ! Input/Output variables.

  ! Base_Structure object.
  type(Base_Structure_type), intent(in) :: Structure

  ! Output variables.
```

```
    ! Base_Structure value to be output.
    type(integer), dimension(NPEs) :: Length_Vector

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Structure),5)       ! Base_Structure is valid.

    ! Set value.

    Length_Vector = Structure%Length_Vector

    ! Verify guarantees - none.

    return
  end function Get_Length_Vector_Structure

  function Get_Locus_Structure (Structure) result(Locus)

    ! Input variables.

    ! Base_Structure object.
    type(Base_Structure_type), intent(in) :: Structure

    ! Output variables.

    type(character,80) :: Locus          ! Base_Structure value to be output.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Structure),5)     ! Base_Structure is valid.

    ! Set value.

    Locus = Structure%Locus

    ! Verify guarantees - none.

    return
  end function Get_Locus_Structure

  function Get_Range_PE_Structure (Structure) result(Range_PE)

    ! Input/Output variables.

    ! Base_Structure object.
    type(Base_Structure_type), intent(in) :: Structure

    ! Output variables.

    ! Base_Structure value to be output.
```

```
    type(integer), dimension(2) :: Range_PE

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Structure),5)       ! Base_Structure is valid.

    ! Set value.

    Range_PE = Structure%Range_PE

    ! Verify guarantees - none.

    return
  end function Get_Range_PE_Structure
```

D.3.8 Output_Base_Structure Procedure

The main documentation of the Output_Base_Structure Procedure in § 9.3.8 on page 83 contains additional explanation of this code listing.

```
  subroutine Output_Base_Structure (Structure, Unit, Type, Indent)

    ! Input variables.

    type(Base_Structure_type), intent(in) :: Structure ! Output Variable.
    type(integer), intent(in), optional :: Unit        ! Output unit.
    type(character,*), optional :: Type                ! Structure type.
    type(integer), optional :: Indent                  ! Indentation.

    ! Internal variables.

    type(integer) :: A_Unit                            ! Actual output unit.
    type(character,80) :: A_Type                       ! Actual structure type.
    type(integer) :: A_Indent                          ! Actual indentation.
    type(integer) :: PE, i                             ! PE loop counter.
    type(character,80) :: Blanks                       ! A line of blanks.
    type(character,80) :: Output_Buffer                ! Output buffer.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Structure),5)      ! Structure is valid.

    ! Set unit number.

    if (PRESENT(Unit)) then
      A_Unit = Unit
    else
      A_Unit = 6
```

```
    end if

    ! Set indentation.

    if (PRESENT(Indent)) then
      A_Indent = Indent
    else
      A_Indent = 0
    end if
    Blanks = ' '

    ! Only output on the IO PE.

    if (this_is_IO_PE) then

      ! Set structure type.

      if (PRESENT(Type)) then
        A_Type = Type
      else
        A_Type = 'Base'
      end if

      ! Output Identification Info.

      write (A_Unit,100) Blanks(1:A_Indent), TRIM(A_Type), &
                         ' Structure Information:'
      write (A_Unit,101) Blanks(1:A_Indent+2), 'Locus               = ', &
                         TRIM(Structure%Locus)
      write (A_Unit,102) Blanks(1:A_Indent+2), 'Initialized         = ', &
                         Initialized(Structure)
      write (A_Unit,103) Blanks(1:A_Indent+2), 'Length_Total        =', &
                         Structure%Length_Total
      if (NPEs <= 4) then
        write (A_Unit,103) Blanks(1:A_Indent+2), 'Length_Vector       =', &
                           (Structure%Length_Vector(PE), PE = 1, MIN(NPEs, 4))
      else
        write (A_Unit,103) Blanks(1:A_Indent+2), 'Length_Vector       =', &
                           (Structure%Length_Vector(PE), PE = 1, 4), ','
        do PE = 5, NPEs, 4
          if (PE + 4 <= NPEs) then
            write (A_Unit,104) Blanks(1:A_Indent+23), &
                               (Structure%Length_Vector(i), &
                               i = PE, MIN(PE+3, NPEs)), ','
          else
            write (A_Unit,104) Blanks(1:A_Indent+23), &
                               (Structure%Length_Vector(i), &
                               i = PE, MIN(PE+3, NPEs))
          end if
        end do
      end if
    end if

    ! PE-dependent output.
```

```
    write (Output_Buffer,105) Blanks(1:A_Indent+2), 'PE:', this_PE, &
                              ', Length_PE =', Structure%Length_PE, &
                              ', Range_PE = (', Structure%Range_PE, ')'
    call Parallel_Write (Output_Buffer, A_Unit)

    ! Format statements. With these formats, this should work up to
    ! (10^6 - 1) PEs and (10^12 - 1) items / PE.

100 format (/, 3a, /)
101 format (3a)
102 format (2a, l2)
103 format (2a, i12, :, 3(',', i12, :), a)
104 format (a, i12, :, 3(',', i12, :),  a)
105 format (2a, i5, a, i12, a, i12, ',', i12, a)

    ! Verify guarantees - none.

    return
  end subroutine Output_Base_Structure
```

D.3.9 Base_Structure Class Unit Test Program

This lightly commented program performs a unit test on the Base_Structure Class, which is described in § 9.3 on page 79.

```
program Unit_Test

  use Caesar_Intrinsics_Module
  use Caesar_Base_Structure_Class
  use Caesar_Communication_Class
  implicit none

  type(Communication_type) :: Comm
  type(Base_Structure_type) :: Cell_Structure
  type(Status_type) :: status
  type(character,name_length) :: Cell_Locus
  type(integer,1) :: Cell_Length_Vector
  type(integer) :: i

  ! Initializations.

  call Initialize (Comm)
  call Output (Comm)
  call Initialize (status)
  call Initialize (Cell_Locus)
  call Initialize (Cell_Length_Vector, NPEs)

  ! Set up.

  Cell_Locus = 'Cells'
  Cell_Length_Vector = (/ (i**2, i = 1, NPEs) /)
```

```
  ! Initialize base structure.

  call Initialize (Cell_Structure, Cell_Length_Vector, Cell_Locus, status)

  ! Output info.

  call Output (Cell_Structure)

  ! Check state of base structure.

  VERIFY(Valid_State(Cell_Structure),0)

  ! Generate an even distribution (only output on error).

  call Generate_Even_Distribution (Cell_Length_Vector, 123456789)

  ! Finalize base structure and communications.

  call Finalize (Cell_Structure)
  call Finalize (Comm)

end
```

D.4 Data_Index Class Code Listing

The main documentation of the Data_Index Class in § 9.4 on page 84 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 09/15/99
! CVS Info:   $Id: data_index.F90,v 8.4 2006/10/17 23:01:53 hall Exp $

module Caesar_Data_Index_Class

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_Trace_Class
  use Caesar_Communication_Class
  use Caesar_Base_Structure_Class
  use Caesar_Numbers_Module, only: zero

  ! Start up with everything untyped and private.

  implicit none
```

```
private

! Public procedures.

public :: Initialize, Finalize, Valid_State, Initialized
public :: Assignment (=), Get_Values, Initialize_Shell_Partition, Output

interface Initialize
  module procedure Initialize_Data_Index
end interface

interface Finalize
  module procedure Finalize_Data_Index
end interface

interface Valid_State
  module procedure Valid_State_Data_Index
end interface

interface Initialized
  module procedure Initialized_Data_Index
end interface

interface Assignment (=)
  module procedure Get_Values_Data_Index_1
  module procedure Get_Values_Data_Index_2
end interface

interface Get_Values
  module procedure Get_Values_Data_Index_1
  module procedure Get_Values_Data_Index_2
end interface

interface Output
  module procedure Output_Data_Index
end interface

! Public type definitions.

public :: Data_Index_type

type Data_Index_type

  ! Initialization flag.

  type(integer) :: Initialized

  ! Basic data structures. The Many_Structure corresponds to the columns in
  ! the index array. The One_Structure corresponds to the rows in the index
  ! array. The index array can be thought of as a "Many of One" relationship
  ! (e.g. Many Faces of Each Cell, or Faces_of_Cells), with each row of the
  ! array signifying all the "Many" items that correspond to that "One" row.

  type(Base_Structure_type), pointer :: Many_Structure
```

```
    type(Base_Structure_type), pointer :: One_Structure

    ! The number of dimensions that the index has. "Ragged_Right" indices
    ! are signified by a Dimensionality of -1, and are equivalent to a
    ! Dimensionality of 2 where the number of columns per row varies.

    type(integer) :: Dimensionality

    ! The index values, which are modified: 1. to reflect off-PE locations
    ! with a negative number corresponding to the location in Off_PE_Index
    ! of the original value; and 2. to have a local numbering instead of a
    ! global numbering.

    type(integer,1) :: Index1
    type(integer,2) :: Index2
    ! Needed for polyhedral "Faces of Cells", "Nodes of Faces".
    ! type(Ragged_Integer_type)  :: IndexRR

    ! The trace for the communication associated with the index.

    type(Trace_type) :: Trace

    ! The values of the index which are not local to this PE.

    type(integer,1) :: Off_PE_Index

    ! The number of Off-PE values.

    type(integer) :: NOff_PE

    ! The trace for the communication associated with the Off_PE_Index.

    type(Trace_type) :: Off_PE_Trace

  end type Data_Index_type

contains
```

The Data_Index Class contains the following routines which are listed in separate sections:

end module Caesar_Data_Index_Class

D.4.1 Initialize_Data_Index Procedure

The main documentation of the Initialize_Data_Index Procedure in § 9.4.1 on page 85 contains additional explanation of this code listing.

```
subroutine Initialize_Data_Index (Index, Many_Structure, One_Structure, &
                                  Many_of_One_Vector, Many_of_One_Array, &
                                  ! Many_of_One_Ragged, &
                                  status)

  ! Use associations.

  use Caesar_Flags_Module, only: initialized_flag

  ! Input variables.

  type(Base_Structure_type), target :: Many_Structure ! Column base structure.
  type(Base_Structure_type), target :: One_Structure  ! Row base structure.
  type(integer,1), optional :: Many_of_One_Vector     ! Vector indices.
  type(integer,2), optional :: Many_of_One_Array      ! Array indices.
  !type(Ragged_Integer_type), optional :: Many_of_One_Ragged ! Ragged indices.

  ! Output variables.

  ! Data_Index to be initialized.
  type(Data_Index_type), intent(out) :: Index
  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(Status_type), dimension(6) :: allocate_status ! Allocation Status.
  type(Status_type) :: consolidated_status        ! Consolidated Status.
  type(integer,1) :: Off_PE_Index_Temp            ! Off_PE_Index temporary.
  type(integer) :: entry, i, j                    ! Loop indices.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Many_Structure),5) ! Many_Structure is valid.
  VERIFY(Valid_State(One_Structure),5)  ! One_Structure is valid.

  ! Set allocation status.

  call Initialize (allocate_status)
  call Initialize (consolidated_status)

  ! Set up trace.

  if (PRESENT(Many_of_One_Vector)) then
    call Initialize (Index%Trace, Many_of_One_Vector, &
                     Length_PE(Many_Structure), allocate_status(1))
  else if (PRESENT(Many_of_One_Array)) then
```

```
      call Initialize (Index%Trace, Many_of_One_Array, &
                       Length_PE(Many_Structure), allocate_status(1))
    end if

    ! Initialize temporary to store off-PE indices.
    !
    ! Note that the size that this variable needs to be is unknown. Assuming
    ! that the number of off-PE indices is less than the number of on-PE
    ! indices gives a size of Length_PE(One_Structure). This assumption
    ! means that the number of "boundary" values needed is less than
    ! the number of on-processor values, which will often be a reasonable
    ! assumption. However, one can imagine an almost worst-case scenario (not
    ! counting an extremely bad poly-connected structure, like a polyhedral
    ! mesh) consisting of a contiguous line of items connected like a
    ! 3-D structured hex mesh. Each item would connect to 8 off-PE items
    ! perpindicularly, and some others that are overlapped. The "end-caps"
    ! would not overlap, so would have an additional 9 items each. This
    ! gives rise to the following formula for the maximum:
    !
    !   Max size = 8 * Length_PE(One_Structure) + 18
    !
    ! This size is used for a temporary -- the actual variable is then sized
    ! to be exactly the needed size.

    call Initialize (Off_PE_Index_Temp, 8*Length_PE(One_Structure) + 18, &
                     allocate_status(2))

    ! Set up structure pointers.

    Index%Many_Structure => Many_Structure
    Index%One_Structure => One_Structure

    ! Set up for a Vector Index.

    if (PRESENT(Many_of_One_Vector)) then

      ! Verifications for a Vector.

      VERIFY(.not.PRESENT(Many_of_One_Array),7)
      VERIFY(.not.PRESENT(Many_of_One_Ragged),1000) ! Activate this later.
      VERIFY(Length_PE(One_Structure) == SIZE(Many_of_One_Vector), 5)

      ! Initialize Index1.

      Index%Dimensionality = 1
      call Initialize (Index%Index1, SIZE(Many_of_One_Vector,1), &
                       allocate_status(3))
      Index%Index1 = Many_of_One_Vector

      ! Set up the Off_PE_Index vector, and modify the Index1 vector to
      ! point off-PE references into Off_PE_Index with a negative flag.
      ! Also, change from global to local numbering.

      ! Loop over Index1 variables.
```

```
    Index%NOff_PE = 0
    do i = 1, SIZE(Many_of_One_Vector,1)

      ! Select Off-PE entries in Index1.

      if (Index%Index1(i) .NotInInterval. Range_PE(Many_Structure)) then

        ! If this value of Index1 hasn't been stored, store it.

        if (.not.ANY(Index%Index1(i) == Off_PE_Index_Temp)) then
          Index%NOff_PE = Index%NOff_PE + 1
          Off_PE_Index_Temp(Index%NOff_PE) = Index%Index1(i)
          Index%Index1(i) = -Index%NOff_PE

        ! Otherwise, figure out which entry in Off_PE_Index_Temp
        ! to set Index1 to (with a negative flag).

        else
          do entry = 1, Index%NOff_PE
            if (Index%Index1(i) == Off_PE_Index_Temp(entry)) then
              Index%Index1(i) = -entry
            end if
          end do
        end if

      ! For the On-PE indices, change to a local numbering.

      else

        Index%Index1(i) = Index%Index1(i) - First_PE(Many_Structure) + 1

      end if
    end do

  ! Set up for an Array Index.

  else if (PRESENT(Many_of_One_Array)) then

    ! Verifications for an Array.

    VERIFY(.not.PRESENT(Many_of_One_Vector),7)
    VERIFY(.not.PRESENT(Many_of_One_Ragged),1000) ! Activate this later.
    VERIFY(Length_PE(One_Structure) == SIZE(Many_of_One_Array,1), 5)

    ! Initialize Index2.

    Index%Dimensionality = 2
    call Initialize (Index%Index2, SIZE(Many_of_One_Array,1), &
                     SIZE(Many_of_One_Array,2), allocate_status(3))
    Index%Index2 = Many_of_One_Array

    ! Set up the Off_PE_Index vector, and modify the Index2 array to
    ! point off-PE references into Off_PE_Index with a negative flag.
```

```
    ! Also, change from global to local numbering.

    ! Loop over Index2 variables.

    Index%NOff_PE = 0
    do i = 1, SIZE(Many_of_One_Array,1)
      do j = 1, SIZE(Many_of_One_Array,2)

        ! Select Off-PE entries in Index2.

        if (Index%Index2(i,j) .NotInInterval. Range_PE(Many_Structure)) then

          ! If this value of Index2 hasn't been stored, store it.

          if (.not.ANY(Index%Index2(i,j) == Off_PE_Index_Temp)) then
            Index%NOff_PE = Index%NOff_PE + 1
            Off_PE_Index_Temp(Index%NOff_PE) = Index%Index2(i,j)
            Index%Index2(i,j) = -Index%NOff_PE

          ! Otherwise, figure out which entry in Off_PE_Index_Temp
          ! to set Index2 to (with a negative flag).

          else
            do entry = 1, Index%NOff_PE
              if (Index%Index2(i,j) == Off_PE_Index_Temp(entry)) then
                Index%Index2(i,j) = -entry
              end if
            end do
          end if

        ! For the On-PE indices, change to a local numbering.

        else

          Index%Index2(i,j) = Index%Index2(i,j) &
                              - First_PE(Many_Structure) + 1

        end if
      end do
    end do

  ! Set up for a Ragged Index.

  ! <in the future>

  end if

  ! Store temporary in final form.

  VERIFY(Index%NOff_PE <= SIZE(Off_PE_Index_Temp),7)
  call Initialize (Index%Off_PE_Index, Index%NOff_PE, allocate_status(4))
  Index%Off_PE_Index = Off_PE_Index_Temp(1:Index%NOff_PE)
  call Finalize (Off_PE_Index_Temp, allocate_status(5))
```

```
    ! Set up off-PE trace.

    !if (Index%NOff_PE /= 0) then
      call Initialize (Index%Off_PE_Trace, Index%Off_PE_Index, &
                       Length_PE(Many_Structure), allocate_status(6))
    !end if

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set initialization flag.

    Index%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(Index),5)  ! Index is now valid.

    return
  end subroutine Initialize_Data_Index
```

D.4.2 Finalize_Data_Index Procedure

The main documentation of the Finalize_Data_Index Procedure in § 9.4.2 on page 85 contains additional explanation of this code listing.

```
  subroutine Finalize_Data_Index (Index, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Data_Index to be finalized.
    type(Data_Index_type), intent(inout) :: Index

    ! Output variable.

    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.
```

```
type(Status_type), dimension(6) :: deallocate_status ! Deallocation Status.
type(Status_type) :: consolidated_status             ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Index),7) ! Index is valid.

! Unset initialization flag.

Index%Initialized = uninitialized_flag

! Set deallocation status.

call Initialize (deallocate_status)
call Initialize (consolidated_status)

! Finalize internals.

NULLIFY(Index%Many_Structure)
NULLIFY(Index%One_Structure)

select case (Index%Dimensionality)
case (1)
  call Finalize (Index%Index1,  deallocate_status(1))
case (2)
  call Finalize (Index%Index2,  deallocate_status(1))
!case (-1)
!  call Finalize (Index%IndexRR, deallocate_status(1))
end select

call Finalize (Index%Dimensionality, deallocate_status(2))
call Finalize (Index%NOff_PE,        deallocate_status(3))
call Finalize (Index%Off_PE_Index,   deallocate_status(4))
if (Initialized(Index%Off_PE_Trace)) then
  call Finalize (Index%Off_PE_Trace, deallocate_status(5))
end if
call Finalize (Index%Trace,          deallocate_status(6))

! Process status variables.

consolidated_status = deallocate_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
  VERIFY(Normal(consolidated_status), 5)
end if
call Finalize (consolidated_status)
call Finalize (deallocate_status)

! Verify guarantees.
```

```
    VERIFY(.not.Valid_State(Index),7) ! Index is not valid.

    return
  end subroutine Finalize_Data_Index
```

D.4.3 Valid_State_Data_Index Procedure

The main documentation of the Valid_State_Data_Index Procedure in § 9.4.3 on page 86 contains additional explanation of this code listing.

```
  function Valid_State_Data_Index (Index) result(Valid)

    ! Input variables.

    ! Variable to be checked.
    type(Data_Index_type), intent(in) :: Index

    ! Output variables.

    type(logical) :: Valid  ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for association of pointered internals.

    Valid = Valid .and. ASSOCIATED(Index%Many_Structure)
    Valid = Valid .and. ASSOCIATED(Index%One_Structure)
    if (.not.Valid) return

    ! Check for validity of internals.

    Valid = Valid .and. Initialized(Index)
    Valid = Valid .and. Valid_State(Index%One_Structure)
    Valid = Valid .and. Valid_State(Index%Many_Structure)
    select case (Index%Dimensionality)
    case (1)
      Valid = Valid .and. Valid_State(Index%Index1)
    case (2)
      Valid = Valid .and. Valid_State(Index%Index2)
    !case (-1)
    !  Valid = Valid .and. Valid_State(Index%IndexRR)
    end select
    Valid = Valid .and. Valid_State(Index%Off_PE_Index)
    if (.not.Valid) return

    ! Checks on the validity of Index.

    Valid = Valid .and. Index%NOff_PE == SIZE(Index%Off_PE_Index)
```

```
    select case (Index%Dimensionality)
    case (1)
      Valid = Valid .and. &
              SIZE(Index%Index1,1) == Length_PE(Index%One_Structure)
    case (2)
      Valid = Valid .and. &
              SIZE(Index%Index2,1) == Length_PE(Index%One_Structure)
    !case (-1)
    !  Valid = Valid .and. &
    !          SIZE(Index%IndexRR,1) == Length_PE(Index%One_Structure)
    end select

    ! All off-PE indices are really off-PE.

    Valid = Valid .and. &
            (Index%Off_PE_Index .NotInInterval. Range_PE(Index%Many_Structure))

    ! All positive indices are on-PE. (Negative ones pass this test also.)

    select case (Index%Dimensionality)
    case (1)
      Valid = Valid .and. &
              (ALL(Index%Index1 <= Length_PE(Index%Many_Structure)))
    case (2)
      Valid = Valid .and. &
              (ALL(Index%Index2 <= Length_PE(Index%Many_Structure)))
    !case (-1)
    !  Valid = Valid .and. &
    !          (ALL((Index%IndexRR <= Last_PE(Index%One_Structure) .and. &
    !                Index%IndexRR >= First_PE(Index%One_Structure)) .or. &
    !                Index%IndexRR < 0))
    end select

    return
  end function Valid_State_Data_Index
```

D.4.4 Initialized_Data_Index Procedure

The main documentation of the Initialized_Data_Index Procedure in § 9.4.4 on page 86 contains additional explanation of this code listing.

```
  function Initialized_Data_Index (Index) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    type(Data_Index_type), intent(in) :: Index  ! Data_Index to be checked.

    ! Output variable.
```

```
    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = Index%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Data_Index
```

D.4.5 Generate_Shell_Partition Procedure

The main documentation of the Generate_Shell_Partition Procedure in § 9.4.5 on page 87 contains additional explanation of this code listing.

```
  subroutine Generate_Shell_Partition (c, i_of_c, j_of_c, k_of_c, &
                                     NDimensions, NNodes_per_Side, Output)

    ! Input variables.

    type(integer), intent(in) :: NDimensions     ! Number of dimensions.
    type(integer), intent(in) :: NNodes_per_Side ! Length of a side.
    type(logical), intent(in) :: Output          ! Output toggle.

    ! Output variables.

    type(integer,3) :: c                       ! Cell numbers for each (i,j,k).
    type(integer,1) :: i_of_c, j_of_c, k_of_c ! i,j,k values for each cell #.

    ! Internal variables.

    type(integer) :: buff_loc             ! Buffer location.
    type(integer) :: i, j, k              ! Loop counters.
    type(integer) :: imax, jmax, kmax     ! Maximum values for i, j, and k.
    type(integer) :: maxij, maxijk        ! Maximum of the current (i,j) or
                                          ! (i,j,k) set.
    type(character,80) :: output_buffer   ! Buffer for output.
    type(logical) :: do_output            ! Output toggle (PE-dependent).

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Determine whether or not to output.

    do_output = this_is_IO_PE .and. Output

    ! Set boundaries.
```

```
    imax = NNodes_per_Side
    if (NDimensions > 1) then
      jmax = imax
    else
      jmax = 1
    end if
    if (NDimensions > 2) then
      kmax = imax
    else
      kmax = 1
    end if

    ! Set cell numbers.

    if (do_output) write (6,'(/,a,/)') 'Shell Partitioning:'

    do k = kmax, 1, -1
      do j = jmax, 1,  -1
        buff_loc = 1
        do i = 1, imax
          maxij = MAX(i,j)
          maxijk = MAX(i,j,k)
          select case (NDimensions)
          case (1)
            c(i,j,k) = i
          case (2)
            c(i,j,k) = i + &
                       (maxij - j) + &
                       (maxijk - 1)**NDimensions
          case (3)
            c(i,j,k) = i + &
                       (maxij - j) + &
                       (maxijk - 1)**NDimensions + &
                       (maxijk - k) * (2*maxijk - 1) + &
                       (maxij - 1)**(NDimensions-1)
          end select
          i_of_c(c(i,j,k)) = i
          j_of_c(c(i,j,k)) = j
          k_of_c(c(i,j,k)) = k
          if (do_output) then
            write (output_buffer(buff_loc:),'(i6)') c(i,j,k)
            buff_loc = buff_loc + 6
            if (buff_loc > 75) then
              write (6,*) output_buffer(1:buff_loc-1)
              buff_loc = 1
            end if
          end if
        end do
        if (do_output .and. buff_loc /= 1) then
          write (6,*) output_buffer(1:buff_loc-1)
        end if
      end do
      if (do_output) write (6,*)
```

```
    end do
    return

  end subroutine Generate_Shell_Partition
```

D.4.6 Get_Values_Data_Index Procedure

The main documentation of the Get_Values_Data_Index Procedure in § 9.4.6 on page 87 contains additional explanation of this code listing.

```
  subroutine Get_Values_Data_Index_1 (Values, Index)

    ! Input variable.

    type(Data_Index_type), intent(in) :: Index   ! Variable to be queried.

    ! Input/Output variable.

    type(integer,1,np), intent(inout) :: Values  ! Values bare naked vector.

    ! Internal variables.

    type(integer) :: i                           ! Loop counter.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Index),5)           ! Index is valid.
    VERIFY(Index%Dimensionality==1,5)      ! Dimensionality is correct.

    ! Calculate the values, correcting for local/global and off-PE conversions.

    Values = zero
    do i = 1, SIZE(Index%Index1,1)
      if (Index%Index1(i) > 0) then
        Values(i) = Index%Index1(i) + First_PE(Index%Many_Structure) - 1
      else if (Index%Index1(i) < 0) then
        Values(i) = Index%Off_PE_Index(-Index%Index1(i))
      end if
    end do

    ! Verify guarantees.

    VERIFY(Valid_State_NP(Values),5)       ! Values is valid.

    return
  end subroutine Get_Values_Data_Index_1

  subroutine Get_Values_Data_Index_2 (Values, Index)

    ! Input variable.
```

```
    type(Data_Index_type), intent(in) :: Index  ! Variable to be queried.

    ! Input/Output variable.

    type(integer,2,np), intent(inout) :: Values ! Values bare naked vector.

    ! Internal variables.

    type(integer) :: i, j                       ! Loop counters.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Index),5)           ! Index is valid.
    VERIFY(Index%Dimensionality==2,5)      ! Dimensionality is correct.

    ! Calculate the values, correcting for local/global and off-PE conversions.

    Values = zero
    do i = 1, SIZE(Index%Index2,1)
      do j = 1, SIZE(Index%Index2,2)
        if (Index%Index2(i,j) > 0) then
          Values(i,j) = Index%Index2(i,j) + First_PE(Index%Many_Structure) - 1
        else if (Index%Index2(i,j) < 0) then
          Values(i,j) = Index%Off_PE_Index(-Index%Index2(i,j))
        end if
      end do
    end do

    ! Verify guarantees.

    VERIFY(Valid_State_NP(Values),5)       ! Values is valid.

    return
  end subroutine Get_Values_Data_Index_2
```

D.4.7 Initialize_Shell_Partition Procedure

The main documentation of the Initialize_Shell_Partition Procedure in § 9.4.7 on page 88 contains additional explanation of this code listing.

```
  subroutine Initialize_Shell_Partition (NDimensions, Cell_Structure, &
                                         Node_Structure, &
                                         Nodes_of_Cells_Index, Output)

    ! Input variables.

    type(integer) :: NDimensions                  ! Number of dimensions.
    type(logical) :: Output                       ! Output toggle.
```

```
! Output variables.

type(Base_Structure_type) :: Cell_Structure   ! Structure for the Cells.
type(Base_Structure_type) :: Node_Structure   ! Structure for the Nodes.
type(Data_Index_type) :: Nodes_of_Cells_Index ! Data Index object.

! Internal variables.

type(integer,3) :: c                      ! Shell Partition numbers.
type(integer) :: cell                     ! Loop counter.
type(integer) :: cell_PE                  ! Cell # on this PE.
type(integer) :: i, j, k                  ! Structured mesh indices.
type(integer,1) :: i_of_c, j_of_c, k_of_c ! Inverse Shell Partition
                                          ! numbers.
type(integer,1) :: Length_Vector          ! Lengths for each PE.
type(character,name_length) :: Locus      ! Locus name.
type(integer) :: NCells_PE                ! Number of cells on this PE.
type(integer) :: NNodes_per_Side          ! Number of nodes of one edge.
type(integer) :: NNodes_per_Cell          ! Local # of nodes per cell.
type(integer) :: NNodes_Total             ! Number of nodes in the whole
                                          ! mesh.
type(integer,2) :: Nodes_of_Cells_Array   ! Index array.
type(Status_type) :: status               ! Status variable.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Initialize temporaries.

call Initialize (status)
call Initialize (Locus)
call Initialize (Length_Vector, NPEs)
NNodes_per_Side = NPEs + 1
call Initialize (c, NNodes_per_Side, NNodes_per_Side, NNodes_per_Side)

! NDimensions-dependent Initializations.

NNodes_Total = NNodes_per_Side**NDimensions
call Initialize (i_of_c, NNodes_Total)
call Initialize (j_of_c, NNodes_Total)
call Initialize (k_of_c, NNodes_Total)

! Set up cell structure (Shell Partitioning).

Locus = 'Cells'
Length_Vector = (/ (i**NDimensions - (i-1)**NDimensions, i = 1, NPEs) /)
call Initialize (Cell_Structure, Length_Vector, Locus, status)

! Set up node structure (Shell Partitioning plus extra layer of nodes).

Locus = 'Nodes'
Length_Vector(NPEs) = Length_Vector(NPEs) + &
                      (NPEs+1)**NDimensions - NPEs**NDimensions
call Initialize (Node_Structure, Length_Vector, Locus, status)
```

```
    ! Generate Shell Partitioning numbers.

    call Generate_Shell_Partition (c, i_of_c, j_of_c, k_of_c, NDimensions, &
                                   NNodes_per_Side, Output)

    ! Set up Nodes_of_Cells array.

    NCells_PE = Length_PE(Cell_Structure)
    NNodes_per_Cell = 2**NDimensions
    call Initialize (Nodes_of_Cells_Array, NCells_PE, NNodes_per_Cell)
    do cell = First_PE(Cell_Structure), Last_PE(Cell_Structure)
      i = i_of_c(cell)
      j = j_of_c(cell)
      k = k_of_c(cell)
      cell_PE = cell - First_PE(Cell_Structure) + 1
      Nodes_of_Cells_Array(cell_PE,1) = c(i,j,k)
      Nodes_of_Cells_Array(cell_PE,2) = c(i+1,j,k)
      if (NDimensions >= 2) then
        Nodes_of_Cells_Array(cell_PE,3) = c(i,j+1,k)
        Nodes_of_Cells_Array(cell_PE,4) = c(i+1,j+1,k)
        if (NDimensions == 3) then
          Nodes_of_Cells_Array(cell_PE,5) = c(i,j,k+1)
          Nodes_of_Cells_Array(cell_PE,6) = c(i+1,j,k+1)
          Nodes_of_Cells_Array(cell_PE,7) = c(i,j+1,k+1)
          Nodes_of_Cells_Array(cell_PE,8) = c(i+1,j+1,k+1)
        end if
      end if
    end do

    ! Set up Nodes_of_Cells data index.

    call Initialize (Nodes_of_Cells_Index, Node_Structure, Cell_Structure, &
                     Many_of_One_Array=Nodes_of_Cells_Array, status=status)

    ! Finalize temporaries.

    call Finalize (Length_Vector)
    call Finalize (c)
    call Finalize (Locus)
    call Finalize (status)

    ! Finalize NDimensions-dependent data structures.

    call Finalize (i_of_c)
    call Finalize (j_of_c)
    call Finalize (k_of_c)
    call Finalize (Nodes_of_Cells_Array)

  end subroutine Initialize_Shell_Partition
```

D.4.8 Output_Data_Index Procedure

The main documentation of the Output_Data_Index Procedure in § 9.4.8 on page 88 contains additional explanation of this code listing.

```
subroutine Output_Data_Index (Index, First, Last, Unit, Indent, Output_OPE)

  ! Input variables.

  type(Data_Index_type), intent(in) :: Index        ! Variable to be output.
  type(integer), intent(in), optional :: First      ! Extents of value data
  type(integer), intent(in), optional :: Last       !   to be output.
  type(integer), intent(in), optional :: Unit       ! Output unit.
  type(integer), intent(in), optional :: Indent     ! Indentation.
  type(logical), intent(in), optional :: Output_OPE ! Output OPE toggle.

  ! Internal variables.

  type(integer) :: Buffer_Loc                     ! Buffer location.
  type(integer) :: Buffer_Size                    ! Output buffer size.
  type(integer) :: Buffer_Skip                    ! Buffer increment.
  type(integer) :: i_global, i_local              ! Loop counters.
  type(integer) :: A_First                        ! Actual first value.
  type(integer) :: A_Last                         ! Actual last value.
  type(integer) :: A_Unit                         ! Actual output unit.
  type(logical) :: A_Output_OPE                   ! Actual output OPE toggle.
  type(integer) :: i, j, OPE                      ! Off-PE loop counters.
  type(integer) :: A_Indent                       ! Actual indentation.
  type(character,80) :: Blanks                    ! A line of blanks.
  type(character,80) :: Output_1                  ! Output buffer.
  type(character,80,1) :: Output_Buffer           ! Output buffer vector.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Index),5)      ! Index is valid.

  ! Set unit number.

  if (PRESENT(Unit)) then
    A_Unit = Unit
  else
    A_Unit = 6
  end if

  ! Set indentation.

  if (PRESENT(Indent)) then
    A_Indent = Indent
  else
    A_Indent = 0
  end if
```

```
Blanks = ' '

! Output Identification Info.

if (this_is_IO_PE) then
  write (A_Unit,100) Blanks(1:A_Indent), 'Data Index Information:'
  write (A_Unit,101) Blanks(1:A_Indent+2), 'Locus                 = ', &
                     TRIM(Locus(Index%Many_Structure)), ' of ', &
                     TRIM(Locus(Index%One_Structure))
  write (A_Unit,102) Blanks(1:A_Indent+2), 'Initialized           = ', &
                     Initialized(Index)
  write (A_Unit,103) Blanks(1:A_Indent+2), 'Dimensionality         =', &
                     Index%Dimensionality
end if

! PE-dependent info.

write (Output_1,104) Blanks(1:A_Indent+2), 'PE:', this_PE, &
                     ', NOff_PE      =', Index%NOff_PE
call Parallel_Write (Output_1, A_Unit)

if (PRESENT(Output_OPE)) then
  A_Output_OPE = Output_OPE
else
  A_Output_OPE = .true.
end if
if (A_Output_OPE) then
  Buffer_Size = MAX(1, (SIZE(Index%Off_PE_Index) + 3)/ 4)
  call Initialize (Output_Buffer, Buffer_Size)

  if (Index%NOff_PE <= 4) then
    write (Output_Buffer,105) Blanks(1:A_Indent+2), 'PE:', this_PE, &
                              ', Off_PE_Index =', &
                              (Index%Off_PE_Index(OPE), &
                              OPE = 1, MIN(Index%NOff_PE, 4))
  else
    write (Output_Buffer,105) Blanks(1:A_Indent+2), 'PE:', this_PE, &
                              ', Off_PE_Index =', &
                              (Index%Off_PE_Index(OPE), OPE = 1, 4), ','
    j = 2
    do OPE = 5, Index%NOff_PE, 4
      if (OPE + 4 <= Index%NOff_PE) then
        write (Output_Buffer(j),106) Blanks(1:A_Indent+26), &
                                     (Index%Off_PE_Index(i), &
                                     i = OPE, MIN(OPE+3, Index%NOff_PE)), &
                                     ','
      else
        write (Output_Buffer(j),106) Blanks(1:A_Indent+26), &
                                     (Index%Off_PE_Index(i), &
                                     i = OPE, MIN(OPE+3, Index%NOff_PE))
      end if
      j = j+1
    end do
  end if
```

```
  call Parallel_Write (Output_Buffer, A_Unit)
  call Finalize (Output_Buffer)

end if

! Output internal structure info.

call Output (Index%Many_Structure, A_Unit, 'Many', A_Indent+2)
call Output (Index%One_Structure, A_Unit, 'One', A_Indent+2)
!call Output (Index%Trace, A_Unit)
!call Output (Index%Off_PE_Trace, A_Unit)

! Output internal values.

if (this_is_IO_PE) then
  write (A_Unit,100) Blanks(1:A_Indent), '  Internal Values:'
end if

! Set up local limits in terms of global limits.

if (PRESENT(First)) then
  A_First = First
else
  A_First = 1
end if
if (PRESENT(Last)) then
  A_Last = Last
else
  A_Last = Length_Total(Index%One_Structure)
end if
A_First = MAX(A_First, First_PE(Index%One_Structure))
A_Last = MIN(A_Last, Last_PE(Index%One_Structure))

! Output the indices based on the dimensionality.

select case (Index%Dimensionality)
case (1)
  Buffer_Size = MAX(0, (A_Last - A_First + 1))
  call Initialize (Output_Buffer, Buffer_Size)
  if (Buffer_Size /= 0) then
    Buffer_Skip = 1
    Buffer_Loc = 1
    do i_global = A_First, A_Last
      i_local = i_global - First_PE(Index%One_Structure) + 1
      write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),107) &
        'PE:', this_PE, ', Index1(', i_global, ') =', &
        Index%Index1(i_local)
      Buffer_Loc = Buffer_Loc + Buffer_Skip
    end do
  end if
case (2)
  Buffer_Size = MAX(0, ((SIZE(Index%Index2(1,:)) + 2) / 3) &
                * (A_Last - A_First + 1))
```

```
      call Initialize (Output_Buffer, Buffer_Size)
      if (Buffer_Size /= 0) then
        Buffer_Skip = (SIZE(Index%Index2(1,:)) + 2) / 3
        Buffer_Loc = 1
        do i_global = A_First, A_Last
          i_local = i_global - First_PE(Index%One_Structure) + 1
          write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),107) &
            'PE:', this_PE, ', Index2(', i_global, ',:) =', &
            Index%Index2(i_local,:)
          Buffer_Loc = Buffer_Loc + Buffer_Skip
        end do
      end if
!    case (-1)
!      Buffer_Size = MAX(0, ((SIZE(Index%IndexRR(:,1)) + 2) / 3) &
!                    * (A_Last - A_First + 1))
!      call Initialize (Output_Buffer, Buffer_Size)
!      if (Buffer_Size /= 0) then
!        Buffer_Skip = (SIZE(Index%IndexRR(:,1)) + 2) / 3
!        Buffer_Loc = 1
!        do i_global = A_First, A_Last
!          i_local = i_global - First_PE(Index%Structure) + 1
!          write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),107) &
!            'PE:', this_PE, ', IndexRR(:,', i_global, ') =', &
!            Index%IndexRR(:,i_local)
!          Buffer_Loc = Buffer_Loc + Buffer_Skip
!        end do
!      end if
    end select

    ! Add indentation.

    do Buffer_loc = 1, Buffer_Size
      Output_1 = Output_Buffer(Buffer_loc)
      Output_Buffer(Buffer_loc) = Blanks(1:A_Indent) // Output_1
    end do

    call Parallel_Write (Output_Buffer, A_Unit)
    call Finalize (Output_Buffer)

    ! Format statements. With these formats, this should work up to
    ! (10^6 - 1) PEs.

100 format (/, 2a, /)
101 format (5a)
102 format (2a, l2)
103 format (2a, i11)
104 format (2a, i5, a, i11, :, 4(',', i11, :))
105 format (a, a, i5, a, i11, :, 3(',', i11, :), a)
106 format (a, i11, :, 3(',', i11, :), a)
107 format (2x, a, i5, a, i11, a, i13, :, &
            2(',', i13, :), ',', /, &
            (35x, i13, :, 2(',', i13, :), ','))

    ! Verify guarantees - none.
```

```
    return
  end subroutine Output_Data_Index
```

D.4.9 Data_Index Class Unit Test Program

This lightly commented program performs a unit test on the Data_Index Class, which is described in § 9.4 on page 84.

```
program Unit_Test
  use Caesar_Intrinsics_Module
  use Caesar_Base_Structure_Class
  use Caesar_Data_Index_Class
  use Caesar_Communication_Class
  implicit none

  type(Communication_type) :: Comm
  type(Base_Structure_type) :: Cell_Structure, Node_Structure
  type(Data_Index_type) :: Nodes_of_Cells_Index
  type(integer) :: NDimensions
  type(logical) :: detailed_output

  ! Initializations.

  call Initialize (Comm)
  call Output (Comm)
  detailed_output = NPEs < 16

  ! Loop over 1-D, 2-D and 3-D.

  do NDimensions = 1, 3

    if (this_is_IO_PE) write (6,'(/,a,i1)') 'Number of Dimensions = ', &
                                             NDimensions

    ! Set up the Shell Partition Structures.

    call Initialize_Shell_Partition (NDimensions, Cell_Structure, &
                                     Node_Structure, Nodes_of_Cells_Index, &
                                     detailed_output)

    ! Output Index info.

    call Output (Nodes_of_Cells_Index, &
      MAX(1, Length_Total(Cell_Structure)/10), &
      MIN(Length_Total(Cell_Structure), Length_Total(Cell_Structure)/10+50), &
      Output_OPE=detailed_output)

    ! Check state of data index.

    VERIFY(Valid_State(Nodes_of_Cells_Index),0)
```

```
    ! Finalize NDimensions-dependent data structures.

    call Finalize (Nodes_of_Cells_Index)
    call Finalize (Node_Structure)
    call Finalize (Cell_Structure)

  end do
  if (this_is_IO_PE) write (6,*)

  ! Finalize communications.

  call Finalize (Comm)

end
```

D.5 Assembled_Vector Class Code Listing

The main documentation of the Assembled_Vector Class in § 9.5 on page 89 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 10/04/99
! CVS Info:   $Id: assembled_vector.F90,v 4.4 2006/10/17 23:01:53 hall Exp $

module Caesar_Assembled_Vector_Class

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_Communication_Class
  use Caesar_Base_Structure_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Assignment (=), Get_Values, Locus, Name, Output, Set_Values, &
            Set_Version, Version

  interface Initialize
    module procedure Initialize_Assembled_Vector
  end interface
```

```
interface Finalize
  module procedure Finalize_Assembled_Vector
end interface

interface Valid_State
  module procedure Valid_State_Assembled_Vector
end interface

interface Initialized
  module procedure Initialized_Assembled_Vector
end interface

interface Assignment (=)
  module procedure Get_Values_Assembled_Vector_1
  module procedure Get_Values_Assembled_Vector_2
  module procedure Get_Values_Assembled_Vector_3
  module procedure Get_Values_Assembled_Vector_4
  module procedure Set_Values_Assembled_Vector_1
  module procedure Set_Values_Assembled_Vector_2
  module procedure Set_Values_Assembled_Vector_3
  module procedure Set_Values_Assembled_Vector_4
  module procedure Set_Version_Assembled_Vector
end interface

interface Get_Values
  module procedure Get_Values_Assembled_Vector_1
  module procedure Get_Values_Assembled_Vector_2
  module procedure Get_Values_Assembled_Vector_3
  module procedure Get_Values_Assembled_Vector_4
end interface

interface Locus
  module procedure Get_Locus_Assembled_Vector
end interface

interface Name
  module procedure Get_Name_Assembled_Vector
end interface

interface Output
  module procedure Output_Assembled_Vector
end interface

interface Set_Version
  module procedure Set_Version_Assembled_Vector
end interface

interface Set_Values
  module procedure Set_Values_Assembled_Vector_1
  module procedure Set_Values_Assembled_Vector_2
  module procedure Set_Values_Assembled_Vector_3
  module procedure Set_Values_Assembled_Vector_4
end interface
```

```
interface Version
  module procedure Get_Version_Assembled_Vector
end interface

! Public type definitions.

public :: Assembled_Vector_type

type Assembled_Vector_type

  ! Initialization flag.

  type(integer) :: Initialized

  ! The name for this variable (especially useful in a vector of Assembled
  ! Vectors).

  type(character,name_length) :: Name

  ! Version number which is incremented every time the vector is modified,
  ! or is synced with the version number of a data structure that it
  ! depends on when it is updated.

  type(integer) :: Version

  ! Basic data structure for the axis that is spread over the processors.

  type(Base_Structure_type), pointer :: Structure

  ! The number of dimensions that the "vector" has, including the dimension
  ! that is spread over the processors. "Ragged_Right" vectors are signified
  ! by a Dimensionality of -1.

  type(integer) :: Dimensionality

  ! The extents of the dimensions that the "vector" has, including
  ! the dimension that is spread over the processors, which is last.

  type(integer,1) :: Dimensions

  ! Total number of values in this vector.

  type(integer) :: NValues_Total

  ! Values in the vector, only defined on IO_PE. Values may have either 1,
  ! 2, 3, or 4 dimensions, or be a ragged right array. The last dimension
  ! is always the dimension to be spread across the processors. Only one
  ! of the following variables will be allocated for a given object.

  type(real,1) :: Values1
  type(real,2) :: Values2
  type(real,3) :: Values3
  type(real,4) :: Values4
```

```
    ! Needed for Adaptive Angular Refinement.
    ! type(Ragged_Right_Real_type) :: ValuesRR

  end type Assembled_Vector_type

  ! Global class variables.

  type(integer), parameter :: Version_Increment = 1

contains
```

The Assembled_Vector Class contains the following routines which are listed in separate sections:

end module Caesar_Assembled_Vector_Class

D.5.1 Initialize_Assembled_Vector Procedure

The main documentation of the Initialize_Assembled_Vector Procedure in § 9.5.1 on page 89 contains additional explanation of this code listing.

```
  subroutine Initialize_Assembled_Vector (AV, Structure, Dimensionality, &
                                        Name, status, dim1, dim2, dim3)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variables.

    type(Base_Structure_type), target :: Structure            ! Base structure.
    type(character,*), intent(in), optional :: Name           ! Variable name.
    type(integer), intent(in) :: Dimensionality   ! Dimensionality for this AV.
    type(integer), intent(in), optional :: dim1, dim2, dim3   ! Dimensions.
```

```
    ! Output variables.

    ! Assembled_Vector to be initialized.
    type(Assembled_Vector_type), intent(out) :: AV
    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(2) :: allocate_status ! Allocation Status.
    type(Status_type) :: consolidated_status           ! Consolidated Status.
    type(integer) :: Length_PE                         ! Length on this PE.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Structure),5)              ! Structure is valid.
    VERIFY(Dimensionality .InInterval. (/1,4/),5) ! Dimensionality is in range.
    VERIFY(PRESENT(dim1) .or. Dimensionality == 1,5) ! Proper dimensions exist.
    VERIFY(PRESENT(dim2) .or. Dimensionality <= 2,5) ! Proper dimensions exist.
    VERIFY(PRESENT(dim3) .or. Dimensionality <= 3,5) ! Proper dimensions exist.

    ! Set up structure pointer.

    AV%Structure => Structure

    ! Allocations and initializations.

    call Initialize (allocate_status)
    call Initialize (consolidated_status)

    Length_PE = Length_Total(Structure)
    if (this_is_not_IO_PE) Length_PE = 0

    call Initialize (AV%Dimensions, Dimensionality, allocate_status(1))
    AV%Dimensions(Dimensionality) = Length_PE

    select case (Dimensionality)
    case (1)
      call Initialize (AV%Values1, Length_PE, allocate_status(2))
    case (2)
      call Initialize (AV%Values2, dim1, Length_PE, allocate_status(2))
      AV%Dimensions(1) = dim1
    case (3)
      call Initialize (AV%Values3, dim1, dim2, Length_PE, allocate_status(2))
      AV%Dimensions(1) = dim1
      AV%Dimensions(2) = dim2
    case (4)
      call Initialize (AV%Values4, dim1, dim2, dim3, Length_PE, &
                       allocate_status(2))
      AV%Dimensions(1) = dim1
      AV%Dimensions(2) = dim2
      AV%Dimensions(3) = dim3
```

```
    !case (-1)
    !  call Initialize (AV%ValuesRR, Length_Total(Structure), &
    !                   allocate_status)
    !  AV%NValues_Total = F(Length_Total(Structure))
    end select
    AV%NValues_Total = PRODUCT(AV%Dimensions)

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set up internals.

    if (PRESENT(Name)) AV%Name = Name
    AV%Dimensionality = Dimensionality
    AV%Version = 0

    ! Set initialization flag.

    AV%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(AV),5)  ! AV is now valid.

    return
  end subroutine Initialize_Assembled_Vector
```

D.5.2 Finalize_Assembled_Vector Procedure

The main documentation of the Finalize_Assembled_Vector Procedure in § 9.5.2 on page 90 contains additional explanation of this code listing.

```
  subroutine Finalize_Assembled_Vector (AV, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Assembled_Vector to be finalized.
    type(Assembled_Vector_type), intent(inout) :: AV

    ! Output variable.
```

```
type(Status_type), intent(out), optional :: status   ! Exit status.

! Internal variables.

type(Status_type), dimension(6) :: deallocate_status ! Deallocation Status.
type(Status_type) :: consolidated_status             ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(AV),7) ! AV is valid.

! Unset initialization flag.

AV%Initialized = uninitialized_flag

! Set deallocation status.

call Initialize (deallocate_status)
call Initialize (consolidated_status)

! Finalize internals.

NULLIFY(AV%Structure)
select case (AV%Dimensionality)
case (1)
  call Finalize (AV%Values1, deallocate_status(1))
case (2)
  call Finalize (AV%Values2, deallocate_status(1))
case (3)
  call Finalize (AV%Values3, deallocate_status(1))
case (4)
  call Finalize (AV%Values4, deallocate_status(1))
!case (-1)
!  call Finalize (AV%ValuesRR, deallocate_status(1))
end select
call Finalize (AV%Dimensionality, deallocate_status(2))
call Finalize (AV%Dimensions,     deallocate_status(3))
call Finalize (AV%NValues_Total,  deallocate_status(4))
call Finalize (AV%Name,           deallocate_status(5))
call Finalize (AV%Version,        deallocate_status(6))

! Process status variables.

consolidated_status = deallocate_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
  VERIFY(Normal(consolidated_status), 5)
end if
call Finalize (consolidated_status)
call Finalize (deallocate_status)
```

```
    ! Verify guarantees.

    VERIFY(.not.Valid_State(AV),7) ! AV is not valid.

    return
  end subroutine Finalize_Assembled_Vector
```

D.5.3 Valid_State_Assembled_Vector Procedure

The main documentation of the Valid_State_Assembled_Vector Procedure in § 9.5.3 on page 91 contains additional explanation of this code listing.

```
  function Valid_State_Assembled_Vector (AV) result(Valid)

    ! Input variables.

    ! Variable to be checked.
    type(Assembled_Vector_type), intent(in) :: AV

    ! Output variables.

    type(logical) :: Valid       ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for association of pointered internals.

    Valid = Valid .and. ASSOCIATED(AV%Structure)
    if (.not.Valid) return

    ! Check for validity of internals.

    Valid = Valid .and. Initialized(AV)
    Valid = Valid .and. Valid_State(AV%Dimensionality)
    Valid = Valid .and. Valid_State(AV%Dimensions)
    Valid = Valid .and. Valid_State(AV%NValues_Total)
    Valid = Valid .and. Valid_State(AV%Name)
    Valid = Valid .and. Valid_State(AV%Structure)
    select case (AV%Dimensionality)
    case (1)
      Valid = Valid .and. Valid_State(AV%Values1)
    case (2)
      Valid = Valid .and. Valid_State(AV%Values2)
    case (3)
      Valid = Valid .and. Valid_State(AV%Values3)
    case (4)
      Valid = Valid .and. Valid_State(AV%Values4)
```

```
    !case (-1)
    !  Valid = Valid .and. Valid_State(AV%ValuesRR)
    end select
    Valid = Valid .and. Valid_State(AV%Version)
    if (.not.Valid) return

    ! Checks on the validity of AV.

    select case (AV%Dimensionality)
    case (1)
      Valid = Valid .and. AV%NValues_Total * delta_PE_IO_PE == SIZE(AV%Values1)
      Valid = Valid .and. ALL(AV%Dimensions == SHAPE(AV%Values1))
    case (2)
      Valid = Valid .and. AV%NValues_Total * delta_PE_IO_PE == SIZE(AV%Values2)
      Valid = Valid .and. ALL(AV%Dimensions == SHAPE(AV%Values2))
    case (3)
      Valid = Valid .and. AV%NValues_Total * delta_PE_IO_PE == SIZE(AV%Values3)
      Valid = Valid .and. ALL(AV%Dimensions == SHAPE(AV%Values3))
    case (4)
      Valid = Valid .and. AV%NValues_Total * delta_PE_IO_PE == SIZE(AV%Values4)
      Valid = Valid .and. ALL(AV%Dimensions == SHAPE(AV%Values4))
    !case (-1)
    !  Valid = Valid .and. AV%NValues_Total == SIZE(AV%ValuesRR)
    end select

    return
  end function Valid_State_Assembled_Vector
```

D.5.4 Initialized_Assembled_Vector Procedure

The main documentation of the Initialized_Assembled_Vector Procedure in § 9.5.4 on page 91 contains additional explanation of this code listing.

```
  function Initialized_Assembled_Vector (AV) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! Assembled_Vector to be checked.
    type(Assembled_Vector_type), intent(in) :: AV

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.
```

```
    ! Set initialized boolean.

    Initialized = AV%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Assembled_Vector
```

D.5.5 Get_Locus_Assembled_Vector Procedure

The main documentation of the Get_Locus_Assembled_Vector Procedure in § 9.5.5 on page 91 contains additional explanation of this code listing.

```
  function Get_Locus_Assembled_Vector (AV) result(Locus_AV)

    ! Input variable.

    type(Assembled_Vector_type), intent(in) :: AV   ! Variable to be queried.

    ! Output variable.

    type(character,name_length) :: Locus_AV         ! Locus of AV.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(AV),5)      ! AV is valid.

    ! Set the value.

    Locus_AV = Locus(AV%Structure)

    ! Verify guarantees - none.

    return
  end function Get_Locus_Assembled_Vector
```

D.5.6 Get_Name_Assembled_Vector Procedure

The main documentation of the Get_Name_Assembled_Vector Procedure in § 9.5.6 on page 92 contains additional explanation of this code listing.

```
  function Get_Name_Assembled_Vector (AV) result(Name)

    ! Input variable.

    type(Assembled_Vector_type), intent(in) :: AV   ! Variable to be queried.
```

```
    ! Output variable.

    type(character,name_length) :: Name             ! Name of AV.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(AV),5)      ! AV is valid.

    ! Set the value.

    Name = AV%Name

    ! Verify guarantees - none.

    return
  end function Get_Name_Assembled_Vector
```

D.5.7 Get_Values_Assembled_Vector Procedure

The main documentation of the Get_Values_Assembled_Vector Procedure in § 9.5.7 on page 92 contains additional explanation of this code listing.

```
  define([GET_VALUES_ROUTINE],[
    pushdef([DIM], [$1])
    pushdef([Get_Values_Assembled_Vector_DIM],
      expand(Get_Values_Assembled_Vector_DIM))

    subroutine Get_Values_Assembled_Vector_DIM (Values, AV)

      ! Input variable.

      type(Assembled_Vector_type), intent(in) :: AV ! Variable to be queried.

      ! Input/Output variable.

      type(real,DIM,np), intent(inout) :: Values   ! Values bare naked vector.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(AV),5)                               ! AV is valid.
      VERIFY(Valid_State_NP(Values) .or. this_is_not_IO_PE,5) ! Values is valid.
      ! Values shape check.
      VERIFY(SHAPE(Values) == SHAPE(AV%Values$1) .or. this_is_not_IO_PE,5)
      VERIFY($1 == AV%Dimensionality,5)   ! AV has been set up for this call.

      ! Get the value.
```

```
    if (this_is_IO_PE) Values = AV%Values$1

    ! Verify guarantees.

    VERIFY(Valid_State(AV),5)                            ! AV is still valid.

    return
  end subroutine Get_Values_Assembled_Vector_DIM

  popdef([DIM])
  popdef([Get_Values_Assembled_Vector_DIM])
])

forloop([Dim],[1],[4],[
  GET_VALUES_ROUTINE(Dim)
])
```

D.5.8 Get_Version_Assembled_Vector Procedure

The main documentation of the Get_Version_Assembled_Vector Procedure in § 9.5.8 on page 92 contains additional explanation of this code listing.

```
function Get_Version_Assembled_Vector (AV) result(Version)

  ! Input variable.

  type(Assembled_Vector_type), intent(in) :: AV   ! Variable to be queried.

  ! Output variable.

  type(integer) :: Version                        ! Version number.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(AV),5)      ! AV is valid.

  ! Get the value.

  Version = AV%Version

  ! Verify guarantees - none.

  return
end function Get_Version_Assembled_Vector
```

D.5.9 Output_Assembled_Vector Procedure

The main documentation of the Output_Assembled_Vector Procedure in § 9.5.9 on page 93 contains additional explanation of this code listing.

```
subroutine Output_Assembled_Vector (AV, First, Last, Unit)

  ! Input variables.

  type(Assembled_Vector_type), intent(in) :: AV   ! Variable to be output.
  type(integer), intent(in), optional :: First    ! Extents of value data
  type(integer), intent(in), optional :: Last     !   to be output.
  type(integer), intent(in), optional :: Unit     ! Output unit.

  ! Internal variables.

  type(integer) :: A_Unit                         ! Actual output unit.
  type(integer) :: A_First                        ! Actual first value.
  type(integer) :: A_Last                         ! Actual last value.
  type(integer) :: i                              ! Loop counter.
  type(character,80) :: Name_Name                 ! Name of the AV.
  type(integer) :: Version_Number                 ! Version of the AV.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(AV),5)      ! AV is valid.

  ! Set unit number.

  if (PRESENT(Unit)) then
    A_Unit = Unit
  else
    A_Unit = 6
  end if

  ! These are evaluated on all PEs -- NOT inside an IO PE block -- because
  ! they contain validity checks on AV and thus require global communication.

  Version_Number = Version(AV)
  Name_Name = Name(AV)

  ! Output Identification Info.

  if (this_is_IO_PE) then
    write (A_Unit,100) 'Assembled Vector Information:'
    write (A_Unit,*) ' Name           = ', TRIM(Name_Name)
    write (A_Unit,*) ' Initialized    = ', Initialized(AV)
    write (A_Unit,*) ' Version        = ', Version_Number
    write (A_Unit,*) ' Dimensionality = ', AV%Dimensionality
    write (A_Unit,*) ' Dimensions     = ', AV%Dimensions
    write (A_Unit,*) ' NValues_Total  = ', AV%NValues_Total
```

```
    end if

    ! Output internal structure info.

    call Output (AV%Structure, A_Unit, 'Base', Indent=2)

    ! Output values.

    if (this_is_IO_PE) then

      write (A_Unit,100) '  Internal Values:'

      if (PRESENT(First)) then
        A_First = First
      else
        A_First = 1
      end if
      if (PRESENT(Last)) then
        A_Last = Last
      else
        A_Last = Length_Total(AV%Structure)
      end if

      select case (AV%Dimensionality)
      case (1)
        do i = A_First, A_Last
          write (A_Unit,101) 'Values1(', i, ') = ', AV%Values1(i)
        end do
      case (2)
        do i = A_First, A_Last
          write (A_Unit,101) 'Values2(:,', i, ') = ', AV%Values2(:,i)
        end do
      case (3)
        do i = A_First, A_Last
          write (A_Unit,101) 'Values3(:,:,', i, ') = ', AV%Values3(:,:,i)
        end do
      case (4)
        do i = A_First, A_Last
          write (A_Unit,101) 'Values4(:,:,:,', i, ') = ', AV%Values4(:,:,:,i)
        end do
      !case (-1)
      !  do i = A_First, A_Last
      !    write (A_Unit,*) 'ValuesRR(:,', i, ') = ', AV%ValuesRR(:,i)
      !  end do
      end select
    end if

    ! Format statements. With these formats, this should work up to
    ! (10^6 - 1) PEs.

100 format (/,a,/)
101 format (2x,a,i11,a,1p,e11.3e3,:,3(',',e11.3e3,:),',',/,&
            (27x,e11.3e3,:,3(',',e11.3e3,:),','))
```

```
  ! Verify guarantees - none.

  return
end subroutine Output_Assembled_Vector
```

D.5.10 Set_Values_Assembled_Vector Procedure

The main documentation of the Set_Values_Assembled_Vector Procedure in § 9.5.10 on page 93 contains additional explanation of this code listing.

```
define([SET_VALUES_ROUTINE],[
  pushdef([DIM], [$1])
  pushdef([Set_Values_Assembled_Vector_DIM],
    expand(Set_Values_Assembled_Vector_DIM))

  subroutine Set_Values_Assembled_Vector_DIM (AV, Values)

    ! Input variable.

    type(real,DIM,np), intent(in) :: Values    ! Values bare naked vector.

    ! Input/Output variable.

    type(Assembled_Vector_type), intent(inout) :: AV   ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(AV),5)                               ! AV is valid.
    VERIFY(Valid_State_NP(Values) .or. this_is_not_IO_PE,5) ! Values is valid.
    ! Values shape check.
    VERIFY(SHAPE(Values) == SHAPE(AV%Values$1) .or. this_is_not_IO_PE,5)
    VERIFY($1 == AV%Dimensionality,5)   ! AV has been set up for this call.

    ! Set the value.

    if (this_is_IO_PE) AV%Values$1 = Values

    ! Increment the version number.

    AV%Version = AV%Version + Version_Increment

    ! Verify guarantees.

    VERIFY(Valid_State(AV),5)                            ! AV is still valid.

    return
  end subroutine Set_Values_Assembled_Vector_DIM

  popdef([DIM])
  popdef([Set_Values_Assembled_Vector_DIM])
```

```
  ])

  forloop([Dim],[1],[4],[
    SET_VALUES_ROUTINE(Dim)
  ])
```

D.5.11 Set_Version_Assembled_Vector Procedure

The main documentation of the Set_Version_Assembled_Vector Procedure in § 9.5.11 on page 93 contains additional explanation of this code listing.

```
  subroutine Set_Version_Assembled_Vector (AV, Version)

    ! Input variable.

    type(integer), intent(in) :: Version               ! Version number.

    ! Input/Output variable.

    type(Assembled_Vector_type), intent(inout) :: AV   ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(AV),5)      ! AV is valid.

    ! Set the value.

    AV%Version = Version

    ! Verify guarantees - none.

    return
  end subroutine Set_Version_Assembled_Vector
```

D.5.12 Assembled_Vector Class Unit Test Program

This lightly commented program performs a unit test on the Assembled_Vector Class, which is described in § 9.5 on page 89.

```
program Unit_Test
  use Caesar_Intrinsics_Module
  use Caesar_Base_Structure_Class
  use Caesar_Assembled_Vector_Class
  use Caesar_Communication_Class
  implicit none

  type(Communication_type) :: Comm
```

```
type(Base_Structure_type) :: Cell_Structure
type(Assembled_Vector_type) :: Coordinates_Cells_AV
type(Status_type) :: status
type(character,name_length) :: Locus_Name, Name_Name
type(real,2) :: Coordinates, Processed_Coordinates
type(integer,1) :: Cells_Length_Vector
type(integer) :: c, CellSize, d, Dimensionality, DimSize, i, NCells, &
                 NDimensions
type(logical) :: Success

! Initializations.

call Initialize (Comm)
call Output (Comm)
call Initialize (status)
call Initialize (Locus_Name)
call Initialize (Name_Name)
call Initialize (Cells_Length_Vector, NPEs)
Locus_Name = 'Cells'
Name_Name = 'Coordinates of Cells'
Cells_Length_Vector = (/ (i**2, i = 1, NPEs) /)
Dimensionality = 2
NDimensions = 4

! Set up Coordinates array on IO PE only.

NCells = SUM(Cells_Length_Vector)
if (this_is_IO_PE) then
  DimSize = NDimensions
  CellSize = NCells
else
  DimSize = 0
  CellSize = 0
end if
call Initialize (Coordinates, DimSize, CellSize)
call Initialize (Processed_Coordinates, DimSize, CellSize)
if (this_is_IO_PE) then
  Coordinates = RESHAPE( &
    (/ (( changetype(real,(d + 10*c)), d = 1,NDimensions), c = 1,NCells) /), &
    (/ NDimensions, NCells /) )
end if

! Initialize base structure and assembled vector.

call Initialize (Cell_Structure, Cells_Length_Vector, Locus_Name, status)
call Initialize (Coordinates_Cells_AV, Cell_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)

! Version number check.

Coordinates_Cells_AV = 123
Success = Version(Coordinates_Cells_AV) == 123
call Output_Test ('Version number', Success)
```

```
  ! Send Coordinates into Assembled Vector and back again.

  Coordinates_Cells_AV = Coordinates
  Processed_Coordinates = Coordinates_Cells_AV

  ! Check to see if the round trip had any effect.

  Success = ALL(Processed_Coordinates == Coordinates)
  call Output_Test ('Round trip', Success)

  ! Output a fraction of the AV.

  call Output (Coordinates_Cells_AV, MAX(1, NCells/10), MIN(NCells, NCells/10+50))

  ! Check state of base structure and assembled vector.

  VERIFY(Valid_State(Coordinates_Cells_AV),0)
  VERIFY(Valid_State(Cell_Structure),0)

  ! Finalize assembled vector, base structure and communications.

  call Finalize (Coordinates_Cells_AV)
  call Finalize (Cell_Structure)
  call Finalize (Comm)

end
```

D.6 Distributed_Vector Class Code Listing

The main documentation of the Distributed_Vector Class in § 9.6 on page 94 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 10/18/99
! CVS Info:   $Id: distributed_vector.F90,v 5.5 2006/10/17 23:01:53 hall Exp $

module Caesar_Distributed_Vector_Class

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_Communication_Class
  use Caesar_Base_Structure_Class
  use Caesar_Assembled_Vector_Class

  ! Start up with everything untyped and private.
```

```
implicit none
private

! Public procedures.

public :: Initialize, Finalize, Valid_State, Initialized, Assignment (=)
public :: Get_Values, Locus, Name, Output, Set_Values, &
          Set_Version, Version

interface Initialize
  module procedure Initialize_Distributed_Vector
end interface

interface Finalize
  module procedure Finalize_Distributed_Vector
end interface

interface Valid_State
  module procedure Valid_State_Distributed_Vector
end interface

interface Initialized
  module procedure Initialized_Distributed_Vector
end interface

interface Assemble
  module procedure Assemble_AV_from_DV
end interface

interface Assignment (=)
  module procedure Assemble_AV_from_DV
  module procedure Distribute_AV_to_DV
  module procedure Get_Values_Distributed_Vector_1
  module procedure Get_Values_Distributed_Vector_2
  module procedure Get_Values_Distributed_Vector_3
  module procedure Get_Values_Distributed_Vector_4
  module procedure Set_Values_Distributed_Vector_1
  module procedure Set_Values_Distributed_Vector_2
  module procedure Set_Values_Distributed_Vector_3
  module procedure Set_Values_Distributed_Vector_4
  module procedure Set_Version_Distributed_Vector
end interface

interface Distribute
  module procedure Distribute_AV_to_DV
end interface

interface Get_Values
  module procedure Get_Values_Distributed_Vector_1
  module procedure Get_Values_Distributed_Vector_2
  module procedure Get_Values_Distributed_Vector_3
  module procedure Get_Values_Distributed_Vector_4
end interface
```

```
interface Locus
  module procedure Get_Locus_Distributed_Vector
end interface

interface Name
  module procedure Get_Name_Distributed_Vector
end interface

interface Output
  module procedure Output_Distributed_Vector
end interface

interface Set_Version
  module procedure Set_Version_Distributed_Vector
end interface

interface Set_Values
  module procedure Set_Values_Distributed_Vector_1
  module procedure Set_Values_Distributed_Vector_2
  module procedure Set_Values_Distributed_Vector_3
  module procedure Set_Values_Distributed_Vector_4
end interface

interface Version
  module procedure Get_Version_Distributed_Vector
end interface

! Public type definitions.

public :: Distributed_Vector_type

type Distributed_Vector_type

  ! Initialization flag.

  type(integer) :: Initialized

  ! The name for this variable (especially useful in a vector of Distributed
  ! Vectors).

  type(character,name_length) :: Name

  ! Version number which is incremented every time the vector is modified,
  ! or is synced with the version number of a data structure that it
  ! depends on when it is updated.

  type(integer) :: Version

  ! Basic data structure for the axis that is spread over the processors.

  type(Base_Structure_type), pointer :: Structure

  ! The number of dimensions that the "vector" has, including the dimension
```

```
    ! that is spread over the processors. "Ragged_Right" vectors are signified
    ! by a Dimensionality of -1.

    type(integer) :: Dimensionality

    ! The extents of the dimensions that the "vector" has, including
    ! the dimension that is spread over the processors, which is last.

    type(integer,1) :: Dimensions

    ! Total number of values in the entire vector (including all PEs).

    type(integer) :: NValues_Total

    ! Number of values on this PE.

    type(integer) :: NValues_PE

    ! A vector containing the number of values on each PE.

    type(integer,1) :: NValues_Vector

    ! Values in the vector, with a different length on each PE. Values may
    ! have either 1, 2, 3, or 4 dimensions, or be a ragged right array. The
    ! last dimension is always the dimension that is spread across the
    ! processors. Only one of the following variables will be allocated for
    ! a given object.

    type(real,1) :: Values1
    type(real,2) :: Values2
    type(real,3) :: Values3
    type(real,4) :: Values4
    ! Needed for Adaptive Angular Refinement.
    ! type(Ragged_Right_Real_type) :: ValuesRR

  end type Distributed_Vector_type

  ! Global class variables.

  type(integer), parameter :: Version_Increment = 4

 ! Not yet implemented:
 !
 ! call Combine_DV_from_BNA (Distributed_Vector, Bare_Naked_Array) --> =  (assume SUM)
 ! (May not be able to make assignment interface due to conflict with BNV accesses.)
 !
 ! Bare_Naked_Array = Gather_Collect_Access (Distributed_Vector, One_Structure, &
 !                    Many_of_One_Index)

contains
```

The Distributed_Vector Class contains the following routines which are listed in separate sections:

end module Caesar_Distributed_Vector_Class

D.6.1 Initialize_Distributed_Vector Procedure

The main documentation of the Initialize_Distributed_Vector Procedure in § 9.6.1 on page 95 contains additional explanation of this code listing.

```
subroutine Initialize_Distributed_Vector (DV, Structure, Dimensionality, &
                                          Name, status, dim1, dim2, dim3)

  ! Use associations.

  use Caesar_Flags_Module, only: initialized_flag

  ! Input variables.

  type(Base_Structure_type), target :: Structure            ! Base structure.
  type(character,*), intent(in), optional :: Name           ! Variable name.
  type(integer), intent(in) :: Dimensionality   ! Dimensionality for this DV.
  type(integer), intent(in), optional :: dim1, dim2, dim3   ! Dimensions.

  ! Output variables.

  ! Distributed_Vector to be initialized.
  type(Distributed_Vector_type), intent(out) :: DV
  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(Status_type), dimension(3) :: allocate_status ! Allocation Status.
  type(Status_type) :: consolidated_status           ! Consolidated Status.
  type(integer) :: NSlice             ! Number of values that are on one
```

```
                                      ! slice, given by a constant location
                                      ! on the distributed axis.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Structure),5)              ! Structure is valid.
VERIFY(Dimensionality .InInterval. (/1,4/),5) ! Dimensionality is in range.
VERIFY(PRESENT(dim1) .or. Dimensionality == 1,5) ! Proper dimensions exist.
VERIFY(PRESENT(dim2) .or. Dimensionality <= 2,5) ! Proper dimensions exist.
VERIFY(PRESENT(dim3) .or. Dimensionality <= 3,5) ! Proper dimensions exist.

! Set up structure pointer.

DV%Structure => Structure

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)

call Initialize (DV%NValues_Vector, NPEs, allocate_status(1))

! Set to maximum dimensionality of 4 to enable VERIFYs without if-checks.
! Also, this gets around problems with using the dimensionality when it
! is set to a negative number for a ragged right array.
call Initialize (DV%Dimensions, 4, allocate_status(2))
DV%Dimensions(Dimensionality) = Length_PE(Structure)

select case (Dimensionality)
case (1)
  call Initialize (DV%Values1, Length_PE(Structure), allocate_status(3))
case (2)
  call Initialize (DV%Values2, dim1, Length_PE(Structure), &
                   allocate_status(3))
  DV%Dimensions(1) = dim1
case (3)
  call Initialize (DV%Values3, dim1, dim2, Length_PE(Structure), &
                   allocate_status(3))
  DV%Dimensions(1) = dim1
  DV%Dimensions(2) = dim2
case (4)
  call Initialize (DV%Values4, dim1, dim2, dim3, Length_PE(Structure), &
                   allocate_status(3))
  DV%Dimensions(1) = dim1
  DV%Dimensions(2) = dim2
  DV%Dimensions(3) = dim3
!case (-1)
!  call Initialize (DV%ValuesRR, Length_PE(Structure), &
!                    allocate_status)
!  DV%NValues_Total = F(Length_Total(Structure))
end select
NSlice = PRODUCT(DV%Dimensions(1:Dimensionality-1))
```

```
    ! Set NValues numbers. (This won't work for Ragged Right arrays --
    ! where the number of values is a function of the length axis.)

    DV%NValues_Total  = NSlice * Length_Total(Structure)
    DV%NValues_PE     = NSlice * Length_PE(Structure)
    DV%NValues_Vector = NSlice * Length_Vector(Structure)

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set up internals.

    if (PRESENT(Name)) DV%Name = Name
    DV%Dimensionality = Dimensionality
    DV%Version = 0

    ! Set initialization flag.

    DV%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(DV),5)  ! DV is now valid.

    return
  end subroutine Initialize_Distributed_Vector
```

D.6.2 Finalize_Distributed_Vector Procedure

The main documentation of the Finalize_Distributed_Vector Procedure in § 9.6.2 on page 96 contains additional explanation of this code listing.

```
  subroutine Finalize_Distributed_Vector (DV, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Distributed_Vector to be finalized.
    type(Distributed_Vector_type), intent(inout) :: DV

    ! Output variable.
```

```
    type(Status_type), intent(out), optional :: status   ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(8) :: deallocate_status ! Deallocation Status.
    type(Status_type) :: consolidated_status             ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(DV),7) ! DV is valid.

    ! Unset initialization flag.

    DV%Initialized = uninitialized_flag

    ! Set deallocation status.

    call Initialize (deallocate_status)
    call Initialize (consolidated_status)

    ! Finalize internals.

    NULLIFY(DV%Structure)
    select case (DV%Dimensionality)
    case (1)
      call Finalize (DV%Values1, deallocate_status(1))
    case (2)
      call Finalize (DV%Values2, deallocate_status(1))
    case (3)
      call Finalize (DV%Values3, deallocate_status(1))
    case (4)
      call Finalize (DV%Values4, deallocate_status(1))
    !case (-1)
    !  call Finalize (DV%ValuesRR, deallocate_status(1))
    end select
    call Finalize (DV%Dimensionality, deallocate_status(2))
    call Finalize (DV%Dimensions,     deallocate_status(3))
    call Finalize (DV%NValues_PE,     deallocate_status(4))
    call Finalize (DV%NValues_Total,  deallocate_status(5))
    call Finalize (DV%NValues_Vector, deallocate_status(6))
    call Finalize (DV%Name,           deallocate_status(7))
    call Finalize (DV%Version,        deallocate_status(8))

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
```

```
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(DV),7) ! DV is not valid.

    return
  end subroutine Finalize_Distributed_Vector
```

D.6.3 Valid_State_Distributed_Vector Procedure

The main documentation of the Valid_State_Distributed_Vector Procedure in § 9.6.3 on page 96 contains additional explanation of this code listing.

```
  function Valid_State_Distributed_Vector (DV) result(Valid)

    ! Input variables.

    ! Variable to be checked.
    type(Distributed_Vector_type), intent(in) :: DV

    ! Output variables.

    type(logical) :: Valid       ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for association of pointered internals.

    Valid = Valid .and. ASSOCIATED(DV%Structure)
    Valid = Valid .and. ASSOCIATED(DV%Dimensions)
    if (.not.Valid) return

    ! Check for validity of internals.

    Valid = Valid .and. Initialized(DV)
    Valid = Valid .and. Valid_State(DV%Dimensionality)
    Valid = Valid .and. Valid_State(DV%Dimensions)
    Valid = Valid .and. Valid_State(DV%NValues_PE)
    Valid = Valid .and. Valid_State(DV%NValues_Total)
    Valid = Valid .and. Valid_State(DV%NValues_Vector)
    Valid = Valid .and. Valid_State(DV%Name)
    Valid = Valid .and. Valid_State(DV%Structure)
    select case (DV%Dimensionality)
    case (1)
      Valid = Valid .and. Valid_State(DV%Values1)
```

```
    case (2)
      Valid = Valid .and. Valid_State(DV%Values2)
    case (3)
      Valid = Valid .and. Valid_State(DV%Values3)
    case (4)
      Valid = Valid .and. Valid_State(DV%Values4)
    !case (-1)
    !  Valid = Valid .and. Valid_State(DV%ValuesRR)
    end select
    Valid = Valid .and. Valid_State(DV%Version)
    if (.not.Valid) return

    ! Checks on the validity of DV.

    select case (DV%Dimensionality)
    case (1)
      Valid = Valid .and. DV%NValues_PE == SIZE(DV%Values1)
      Valid = Valid .and. ALL(DV%Dimensions(1:1) == SHAPE(DV%Values1))
    case (2)
      Valid = Valid .and. DV%NValues_PE == SIZE(DV%Values2)
      Valid = Valid .and. ALL(DV%Dimensions(1:2) == SHAPE(DV%Values2))
    case (3)
      Valid = Valid .and. DV%NValues_PE == SIZE(DV%Values3)
      Valid = Valid .and. ALL(DV%Dimensions(1:3) == SHAPE(DV%Values3))
    case (4)
      Valid = Valid .and. DV%NValues_PE == SIZE(DV%Values4)
      Valid = Valid .and. ALL(DV%Dimensions(1:4) == SHAPE(DV%Values4))
    !case (-1)
    !  Valid = Valid .and. DV%NValues_PE == SIZE(DV%ValuesRR)
    end select

    Valid = Valid .and. DV%NValues_PE == DV%NValues_Vector(this_PE)

    return
  end function Valid_State_Distributed_Vector
```

D.6.4 Initialized_Distributed_Vector Procedure

The main documentation of the Initialized_Distributed_Vector Procedure in § 9.6.4 on page 97 contains additional explanation of this code listing.

```
  function Initialized_Distributed_Vector (DV) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! Distributed_Vector to be checked.
    type(Distributed_Vector_type), intent(in) :: DV
```

```
    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = DV%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Distributed_Vector
```

D.6.5 Assemble_AV_from_DV Procedure

The main documentation of the Assemble_AV_from_DV Procedure in § 9.6.5 on page 97 contains additional explanation of this code listing.

```
  subroutine Assemble_AV_from_DV (AV, DV)

    ! Input variable.

    type(Distributed_Vector_type), intent(in) :: DV  ! DV to be assembled.

    ! Input/Output variable.

    type(Assembled_Vector_type), intent(inout) :: AV ! Assembled vector.

    ! Internal variables.

    type(integer) :: i, j, k  ! Loop counters.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(DV),5)                     ! DV is valid.
    VERIFY(Valid_State(AV),5)                     ! AV is valid.
    VERIFY(ASSOCIATED(AV%Structure,DV%Structure),5) ! AV, DV -> same structure.
    VERIFY(AV%Dimensionality == DV%Dimensionality,5) ! AV, DV have same Dims.

    ! Do the assembly. This could be done faster for multiple dimensions by
    ! packing everything into a single vector, assembling, and then unpacking.
    ! Maybe in a later version.

    select case (DV%Dimensionality)
    case (1)
      call Assemble (AV%Values1, DV%Values1)
```

```
    case (2)
      do i = 1, DV%Dimensions(1)
        call Assemble (AV%Values2(i,:), DV%Values2(i,:))
      end do
    case (3)
      do i = 1, DV%Dimensions(1)
        do j = 1, DV%Dimensions(2)
          call Assemble (AV%Values3(i,j,:), DV%Values3(i,j,:))
        end do
      end do
    case (4)
      do i = 1, DV%Dimensions(1)
        do j = 1, DV%Dimensions(2)
          do k = 1, DV%Dimensions(3)
            call Assemble (AV%Values4(i,j,k,:), DV%Values4(i,j,k,:))
          end do
        end do
      end do
    !case (-1)
    !  call Assemble (AV%ValuesRR, DV%ValuesRR)
    end select

    ! Set the version number.

    AV = Version(DV)

    ! Verify guarantees.

    VERIFY(Valid_State(AV),5)  ! AV is valid.

    return
  end subroutine Assemble_AV_from_DV
```

D.6.6 Distribute_AV_to_DV Procedure

The main documentation of the Distribute_AV_to_DV Procedure in § 9.6.6 on page 97 contains additional explanation of this code listing.

```
  subroutine Distribute_AV_to_DV (DV, AV)

    ! Input variable.

    type(Assembled_Vector_type), intent(in) :: AV      ! AV to be distributed.

    ! Input/Output variable.

    type(Distributed_Vector_type), intent(inout) :: DV ! Distributed vector.

    ! Internal variables.

    type(integer) :: i, j, k  ! Loop counters.
```

```
  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(DV),5)                      ! DV is valid.
  VERIFY(Valid_State(AV),5)                      ! AV is valid.
  VERIFY(ASSOCIATED(AV%Structure,DV%Structure),5) ! AV, DV -> same structure.
  VERIFY(AV%Dimensionality == DV%Dimensionality,5) ! AV, DV have same Dims.

  ! Do the distribution. This could be done faster for multiple dimensions
  ! by packing everything into a single vector, assembling, and then
  ! unpacking. Maybe in a later version.

  select case (DV%Dimensionality)
  case (1)
    call Distribute (DV%Values1, AV%Values1, Length_Vector(AV%Structure))
  case (2)
    do i = 1, DV%Dimensions(1)
      call Distribute (DV%Values2(i,:), AV%Values2(i,:), &
                       Length_Vector(AV%Structure))
    end do
  case (3)
    do i = 1, DV%Dimensions(1)
      do j = 1, DV%Dimensions(2)
        call Distribute (DV%Values3(i,j,:), AV%Values3(i,j,:), &
                         Length_Vector(AV%Structure))
      end do
    end do
  case (4)
    do i = 1, DV%Dimensions(1)
      do j = 1, DV%Dimensions(2)
        do k = 1, DV%Dimensions(3)
          call Distribute (DV%Values4(i,j,k,:), AV%Values4(i,j,k,:), &
                           Length_Vector(AV%Structure))
        end do
      end do
    end do
  !case (-1)
  !  call Distribute (DV%ValuesRR, AV%ValuesRR)
  end select

  ! Set the version number.

  DV = Version(AV)

  ! Verify guarantees.

  VERIFY(Valid_State(DV),5)  ! DV is valid.

  return
end subroutine Distribute_AV_to_DV
```

D.6.7 Get_Locus_Distributed_Vector Procedure

The main documentation of the Get_Locus_Distributed_Vector Procedure in § 9.6.7 on page 98 contains additional explanation of this code listing.

```
function Get_Locus_Distributed_Vector (DV) result(Locus_DV)

  ! Input variable.

  type(Distributed_Vector_type), intent(in) :: DV  ! Variable to be queried.

  ! Output variable.

  type(character,name_length) :: Locus_DV          ! Locus of DV.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(DV),5)      ! DV is valid.

  ! Set the value.

  Locus_DV = Locus(DV%Structure)

  ! Verify guarantees - none.

  return
end function Get_Locus_Distributed_Vector
```

D.6.8 Get_Name_Distributed_Vector Procedure

The main documentation of the Get_Name_Distributed_Vector Procedure in § 9.6.8 on page 98 contains additional explanation of this code listing.

```
function Get_Name_Distributed_Vector (DV) result(Name)

  ! Input variable.

  type(Distributed_Vector_type), intent(in) :: DV ! Variable to be queried.

  ! Output variable.

  type(character,name_length) :: Name             ! Name of DV.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(DV),5)      ! DV is valid.
```

```
  ! Set the value.

  Name = DV%Name

  ! Verify guarantees - none.

  return
end function Get_Name_Distributed_Vector
```

D.6.9 Get_Values_Distributed_Vector Procedure

The main documentation of the Get_Values_Distributed_Vector Procedure in § 9.6.9 on page 98 contains additional explanation of this code listing.

```
define([GET_VALUES_ROUTINE],[
  pushdef([DIM], [$1])
  pushdef([Get_Values_Distributed_Vector_DIM],
    expand(Get_Values_Distributed_Vector_DIM))

  subroutine Get_Values_Distributed_Vector_DIM (Values, DV)

    ! Input variable.

    type(Distributed_Vector_type), intent(in) :: DV ! Variable to be queried.

    ! Input/Output variable.

    type(real,DIM,np), intent(inout) :: Values    ! Values bare naked vector.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(DV),5)                     ! DV is valid.
    VERIFY(Valid_State_NP(Values),5)              ! Values is valid.
    VERIFY(SHAPE(Values) == SHAPE(DV%Values$1),5)  ! Values shape check.
    VERIFY($1 == DV%Dimensionality,5)   ! DV has been set up for this call.

    ! Get the value.

    Values = DV%Values$1

    ! Verify guarantees.

    VERIFY(Valid_State(DV),5)                            ! DV is still valid.

    return
  end subroutine Get_Values_Distributed_Vector_DIM

  popdef([DIM])
  popdef([Get_Values_Distributed_Vector_DIM])
])
```

```
forloop([Dim],[1],[4],[
  GET_VALUES_ROUTINE(Dim)
])
```

D.6.10 Get_Version_Distributed_Vector Procedure

The main documentation of the Get_Version_Distributed_Vector Procedure in § 9.6.10 on page 99 contains additional explanation of this code listing.

```
function Get_Version_Distributed_Vector (DV) result(Version)

  ! Input variable.

  type(Distributed_Vector_type), intent(in) :: DV  ! Variable to be queried.

  ! Output variable.

  type(integer) :: Version                         ! Version number.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(DV),5)      ! DV is valid.

  ! Get the value.

  Version = DV%Version

  ! Verify guarantees - none.

  return
end function Get_Version_Distributed_Vector
```

D.6.11 Output_Distributed_Vector Procedure

The main documentation of the Output_Distributed_Vector Procedure in § 9.6.11 on page 99 contains additional explanation of this code listing.

```
subroutine Output_Distributed_Vector (DV, First, Last, Unit, Indent)

  ! Input variables.

  type(Distributed_Vector_type), intent(in) :: DV ! Variable to be output.
  type(integer), intent(in), optional :: First    ! Extents of value data
  type(integer), intent(in), optional :: Last     !   to be output.
  type(integer), intent(in), optional :: Unit     ! Output unit.
  type(integer), optional :: Indent               ! Indentation.
```

```
! Internal variables.

type(integer) :: Buffer_Loc                    ! Buffer location.
type(integer) :: Buffer_Size                   ! Output buffer size.
type(integer) :: Buffer_Skip                   ! Buffer increment.
type(integer) :: i_global, i_local             ! Loop counters.
type(integer) :: A_First                       ! Actual first value.
type(integer) :: A_Last                        ! Actual last value.
type(integer) :: A_Unit                        ! Actual output unit.
type(integer) :: A_Indent                      ! Actual indentation.
type(integer) :: PE, i                         ! PE loop counter.
type(character,80) :: Blanks                   ! A line of blanks.
type(character,80) :: Name_Name                ! Name of the DV.
type(character,80) :: Output_1                 ! Output buffer.
type(character,80,1) :: Output_Buffer          ! Output buffer vector.
type(integer) :: Version_Number                ! Version of the DV.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(DV),5)      ! DV is valid.

! Set unit number.

if (PRESENT(Unit)) then
  A_Unit = Unit
else
  A_Unit = 6
end if

! Set indentation.

if (PRESENT(Indent)) then
  A_Indent = Indent
else
  A_Indent = 0
end if
Blanks = ' '

! These are evaluated on all PEs -- NOT inside an IO PE block -- because
! they contain validity checks on DV and thus require global communication.

Version_Number = Version(DV)
Name_Name = Name(DV)

! Output Identification Info.

if (this_is_IO_PE) then
  write (A_Unit,100) Blanks(1:A_Indent), 'Distributed Vector Information:'
  write (A_Unit,101) Blanks(1:A_Indent+2), 'Name                 = ', &
                     TRIM(Name_Name)
  write (A_Unit,102) Blanks(1:A_Indent+2), 'Initialized          = ', &
```

```
                     Initialized(DV)
  write (A_Unit,103) Blanks(1:A_Indent+2), 'Version             =', &
                     Version_Number
  write (A_Unit,103) Blanks(1:A_Indent+2), 'Dimensionality      =', &
                     DV%Dimensionality
  write (A_Unit,103) Blanks(1:A_Indent+2), 'NValues_Total       =', &
                     DV%NValues_Total
  !write (A_Unit,101) 'NValues_Vector      =', DV%NValues_Vector

  if (NPEs <= 4) then
    write (A_Unit,103) Blanks(1:A_Indent+2), 'NValues_Vector      =', &
                       (DV%NValues_Vector(PE), PE = 1, MIN(NPEs, 4))
  else
    write (A_Unit,103) Blanks(1:A_Indent+2), 'NValues_Vector      =', &
                       (DV%NValues_Vector(PE), PE = 1, 4), ','
    do PE = 5, NPEs, 4
      if (PE + 4 <= NPEs) then
        write (A_Unit,104) Blanks(1:A_Indent+24), &
                           (DV%NValues_Vector(i), &
                           i = PE, MIN(PE+3, NPEs)), ','
      else
        write (A_Unit,104) Blanks(1:A_Indent+24), &
                           (DV%NValues_Vector(i), &
                           i = PE, MIN(PE+3, NPEs))
      end if
    end do
  end if
end if

! PE-dependent info.

write (Output_1,105) Blanks(1:A_Indent+2), 'PE:', this_PE, &
                     ', Dimensions =', DV%Dimensions
call Parallel_Write (Output_1, A_Unit)
write (Output_1,105) Blanks(1:A_Indent+2), 'PE:', this_PE, &
                     ', NValues_PE =', DV%NValues_PE
call Parallel_Write (Output_1, A_Unit)

! Output internal structure info.

call Output (DV%Structure, A_Unit, 'Base', A_Indent+2)

! Output internal values.

if (this_is_IO_PE) then
  write (A_Unit,100) Blanks(1:A_Indent), '  Internal Values:'
end if

! Set up local limits in terms of global limits.

if (PRESENT(First)) then
  A_First = First
else
  A_First = 1
```

```
end if
if (PRESENT(Last)) then
  A_Last = Last
else
  A_Last = Length_Total(DV%Structure)
end if
A_First = MAX(A_First, First_PE(DV%Structure))
A_Last = MIN(A_Last, Last_PE(DV%Structure))

! Output the values based on the dimensionality.

select case (DV%Dimensionality)
case (1)
  Buffer_Size = MAX(0, (A_Last - A_First + 1))
  call Initialize (Output_Buffer, Buffer_Size)
  if (Buffer_Size /= 0) then
    Buffer_Skip = 1
    Buffer_Loc = 1
    do i_global = A_First, A_Last
      i_local = i_global - First_PE(DV%Structure) + 1
      write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),106) &
        'PE:', this_PE, ', Values1(', i_global, ') =', &
        DV%Values1(i_local)
      Buffer_Loc = Buffer_Loc + Buffer_Skip
    end do
  end if
case (2)
  Buffer_Size = MAX(0, ((SIZE(DV%Values2(:,1)) + 2) / 3) &
              * (A_Last - A_First + 1))
  call Initialize (Output_Buffer, Buffer_Size)
  if (Buffer_Size /= 0) then
    Buffer_Skip = (SIZE(DV%Values2(:,1)) + 2) / 3
    Buffer_Loc = 1
    do i_global = A_First, A_Last
      i_local = i_global - First_PE(DV%Structure) + 1
      write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),106) &
        'PE:', this_PE, ', Values2(:,', i_global, ') =', &
        DV%Values2(:,i_local)
      Buffer_Loc = Buffer_Loc + Buffer_Skip
    end do
  end if
case (3)
  Buffer_Size = MAX(0, ((SIZE(DV%Values3(:,:,1)) + 2) / 3) &
              * (A_Last - A_First + 1))
  call Initialize (Output_Buffer, Buffer_Size)
  if (Buffer_Size /= 0) then
    Buffer_Skip = (SIZE(DV%Values3(:,:,1)) + 2) / 3
    Buffer_Loc = 1
    do i_global = A_First, A_Last
      i_local = i_global - First_PE(DV%Structure) + 1
      write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),106) &
        'PE:', this_PE, ', Values3(:,:,', i_global, ') =', &
        DV%Values3(:,:,i_local)
      Buffer_Loc = Buffer_Loc + Buffer_Skip
```

```
        end do
      end if
    case (4)
      Buffer_Size = MAX(0, ((SIZE(DV%Values4(:,:,:,1)) + 2) / 3) &
                    * (A_Last - A_First + 1))
      call Initialize (Output_Buffer, Buffer_Size)
      if (Buffer_Size /= 0) then
        Buffer_Skip = (SIZE(DV%Values4(:,:,:,1)) + 2) / 3
        Buffer_Loc = 1
        do i_global = A_First, A_Last
          i_local = i_global - First_PE(DV%Structure) + 1
          write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),106) &
            'PE:', this_PE, ', Values4(:,:,:,', i_global, ') =', &
           DV%Values4(:,:,:,i_local)
          Buffer_Loc = Buffer_Loc + Buffer_Skip
        end do
      end if
!    case (-1)
!      Buffer_Size = MAX(0, ((SIZE(DV%ValuesRR(:,1)) + 2) / 3) &
!                    * (A_Last - A_First + 1))
!      call Initialize (Output_Buffer, Buffer_Size)
!      if (Buffer_Size /= 0) then
!        Buffer_Skip = (SIZE(DV%ValuesRR(:,1)) + 2) / 3
!        Buffer_Loc = 1
!        do i_global = A_First, A_Last
!          i_local = i_global - First_PE(DV%Structure) + 1
!          write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),106) &
!            'PE:', this_PE, ', ValuesRR(:,', i_global, ') =', &
!           DV%ValuesRR(:,i_local)
!          Buffer_Loc = Buffer_Loc + Buffer_Skip
!        end do
!      end if
    end select

    ! Add indentation.

    do Buffer_loc = 1, Buffer_Size
      Output_1 = Output_Buffer(Buffer_loc)
      Output_Buffer(Buffer_loc) = Blanks(1:A_Indent) // Output_1
    end do

    call Parallel_Write (Output_Buffer, A_Unit)
    call Finalize (Output_Buffer)

    ! Format statements. With these formats, this should work up to
    ! (10^6 - 1) PEs.

100 format (/, 2a, /)
101 format (3a)
102 format (2a, l2)
103 format (2a, i12, :, 3(',', i12, :), a)
104 format (a, i12, :, 3(',', i12, :), a)
105 format (2a, i5, a, i12, :, 4(',', i12, :))
106 format (2x, a, i5, a, i11, a, 1p, e13.5e3, :, &
```

```
          2(',', e13.5e3, :), ',', /, &
          (36x, e13.5e3, :, 2(',', e13.5e3, :), ','))

    ! Verify guarantees - none.

    return
  end subroutine Output_Distributed_Vector
```

D.6.12 Set_Values_Distributed_Vector Procedure

The main documentation of the Set_Values_Distributed_Vector Procedure in § 9.6.12 on page 99 contains additional explanation of this code listing.

```
  define([SET_VALUES_ROUTINE],[
    pushdef([DIM], [$1])
    pushdef([Set_Values_Distributed_Vector_DIM],
      expand(Set_Values_Distributed_Vector_DIM))

    subroutine Set_Values_Distributed_Vector_DIM (DV, Values)

      ! Input variable.

      type(real,DIM,np), intent(in) :: Values      ! Values bare naked vector.

      ! Input/Output variable.

      type(Distributed_Vector_type), intent(inout) :: DV ! Variable to be set.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(DV),5)                      ! DV is valid.
      VERIFY(Valid_State_NP(Values),5)               ! Values is valid.
      VERIFY($1 == DV%Dimensionality,5)   ! DV has been set up for this call.
      VERIFY(SHAPE(Values) == SHAPE(DV%Values$1),5)  ! Values shape check.

      ! Set the value.

      DV%Values$1 = Values

      ! Increment the version number.

      DV%Version = DV%Version + Version_Increment

      ! Verify guarantees.

      VERIFY(Valid_State(DV),5)                            ! DV is still valid.

      return
    end subroutine Set_Values_Distributed_Vector_DIM
```

```
    popdef([DIM])
    popdef([Set_Values_Distributed_Vector_DIM])
  ])

  forloop([Dim],[1],[4],[
    SET_VALUES_ROUTINE(Dim)
  ])
```

D.6.13 Set_Version_Distributed_Vector Procedure

The main documentation of the Set_Version_Distributed_Vector Procedure in § 9.6.13 on page 100 contains additional explanation of this code listing.

```
  subroutine Set_Version_Distributed_Vector (DV, Version)

    ! Input variable.

    type(integer), intent(in) :: Version                ! Version number.

    ! Input/Output variable.

    type(Distributed_Vector_type), intent(inout) :: DV  ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(DV),5)      ! DV is valid.

    ! Set the value.

    DV%Version = Version

    ! Verify guarantees - none.

    return
  end subroutine Set_Version_Distributed_Vector
```

D.6.14 Distributed_Vector Class Unit Test Program

This lightly commented program performs a unit test on the Distributed_Vector Class, which is described in § 9.6 on page 94.

```
program Unit_Test
  use Caesar_Intrinsics_Module
  use Caesar_Base_Structure_Class
  use Caesar_Assembled_Vector_Class
  use Caesar_Distributed_Vector_Class
  use Caesar_Communication_Class
```

```
implicit none

type(Communication_type) :: Comm
type(Base_Structure_type) :: Cell_Structure
type(Assembled_Vector_type) :: Coordinates_Cells_AV
type(Assembled_Vector_type) :: Processed_Coordinates_Cells_AV
type(Distributed_Vector_type) :: Coordinates_Cells_DV
type(Status_type) :: status
type(character,name_length) :: Locus_Name, Name_Name
type(real,2) :: Coordinates, Processed_Coordinates
type(integer,1) :: Cells_Length_Vector
type(integer) :: c, CellSize, d, Dimensionality, DimSize, i, NCells, &
                 NDimensions
type(logical) :: Success

! Initializations.

call Initialize (Comm)
call Output (Comm)
call Initialize (status)
call Initialize (Locus_Name)
call Initialize (Name_Name)
call Initialize (Cells_Length_Vector, NPEs)
Locus_Name = 'Cells'
Name_Name = 'Coordinates of Cells'
Cells_Length_Vector = (/ (i**2, i = 1, NPEs) /)
Dimensionality = 2
NDimensions = 4

! Set up Coordinates array on IO PE only.

NCells = SUM(Cells_Length_Vector)
if (this_is_IO_PE) then
  DimSize = NDimensions
  CellSize = NCells
else
  DimSize = 0
  CellSize = 0
end if
call Initialize (Coordinates, DimSize, CellSize)
call Initialize (Processed_Coordinates, DimSize, CellSize)
if (this_is_IO_PE) then
  Coordinates = RESHAPE( &
    (/ &
      (( changetype(real,(d + 10*c)), d = 1,NDimensions), c = 1,NCells) &
    /), &
    (/ NDimensions, NCells /) &
  )
end if

! Initialize base structure, assembled vectors, and distributed vector.

call Initialize (Cell_Structure, Cells_Length_Vector, Locus_Name, status)
call Initialize (Coordinates_Cells_AV, Cell_Structure, Dimensionality, &
```

```
                 Name_Name, status, NDimensions)
  call Initialize (Processed_Coordinates_Cells_AV, Cell_Structure, &
                 Dimensionality, Name_Name, status, NDimensions)
  call Initialize (Coordinates_Cells_DV, Cell_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)

  ! Version number check.

  Coordinates_Cells_DV = 123
  Success = Version(Coordinates_Cells_DV) == 123
  call Output_Test ('Version number', Success)

  ! Send Coordinates into Assembled Vector, then Distributed Vector,
  ! and back again.

  Coordinates_Cells_AV = Coordinates
  Coordinates_Cells_DV = Coordinates_Cells_AV
  Processed_Coordinates_Cells_AV = Coordinates_Cells_DV
  Processed_Coordinates = Processed_Coordinates_Cells_AV

  ! Check to see if the round trip had any effect.

  Success = ALL(Processed_Coordinates == Coordinates)
  call Output_Test ('Round trip', Success)

  ! Output the DV.

  call Output (Coordinates_Cells_DV, MAX(1, NCells/10), &
             MIN(NCells, NCells/10+50))

  ! Check state of various objects.

  VERIFY(Valid_State(Coordinates_Cells_AV),0)
  VERIFY(Valid_State(Coordinates_Cells_DV),0)
  VERIFY(Valid_State(Cell_Structure),0)

  ! Finalize assembled vector, base structure and communications.

  call Finalize (Coordinates_Cells_DV)
  call Finalize (Coordinates_Cells_AV)
  call Finalize (Processed_Coordinates_Cells_AV)
  call Finalize (Cell_Structure)
  call Finalize (Comm)

end
```

D.7 Overlapped_Vector Class Code Listing

The main documentation of the Overlapped_Vector Class in § 9.7 on page 100 contains additional explanation of this code listing.

```
!
```

```
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 10/28/99
! CVS Info:   $Id: overlapped_vector.F90,v 9.7 2006/10/17 23:01:53 hall Exp $

module Caesar_Overlapped_Vector_Class

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_Trace_Class
  use Caesar_Communication_Class
  use Caesar_Base_Structure_Class
  use Caesar_Data_Index_Class
  use Caesar_Assembled_Vector_Class
  use Caesar_Distributed_Vector_Class
  use Caesar_Numbers_Module, only: zero

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Assignment (=), Collect_and_Access, Collect_and_Average, &
            Collect_and_MAX, Collect_and_MIN, Collect_and_SUM, Gather, &
            Get_Values, Many_Locus, Name, One_Locus, Output, Set_Version, &
            Version

  interface Initialize
    module procedure Initialize_Overlapped_Vector_1
    module procedure Initialize_Overlapped_Vector_2
  end interface

  interface Finalize
    module procedure Finalize_Overlapped_Vector
  end interface

  interface Valid_State
    module procedure Valid_State_Overlapped_Vector
  end interface

  interface Assignment (=)
    module procedure Collect_and_SUM_DV_from_OV
    module procedure Gather_OV_from_DV
    module procedure Get_Values_Overlapped_Vector_1
    module procedure Get_Values_Overlapped_Vector_2
    module procedure Get_Values_Overlapped_Vector_3
```

```
    module procedure Get_Values_Overlapped_Vector_4
    module procedure Get_Values_Overlapped_Vector_5
    module procedure Set_Version_Overlapped_Vector
  end interface

  interface Collect_and_Access
    module procedure Get_Values_Overlapped_Vector_1
    module procedure Get_Values_Overlapped_Vector_2
    module procedure Get_Values_Overlapped_Vector_3
    module procedure Get_Values_Overlapped_Vector_4
    module procedure Get_Values_Overlapped_Vector_5
  end interface

  fortext([Op],[Average MAX MIN SUM],[
    interface expand(Collect_and_Op)
      module procedure expand(Collect_and_Op_DV_from_OV)
    end interface
  ])

  interface Gather
    module procedure Gather_OV_from_DV
  end interface

  interface Get_Values
    module procedure Get_Values_Overlapped_Vector_1
    module procedure Get_Values_Overlapped_Vector_2
    module procedure Get_Values_Overlapped_Vector_3
    module procedure Get_Values_Overlapped_Vector_4
    module procedure Get_Values_Overlapped_Vector_5
  end interface

  interface Initialized
    module procedure Initialized_Overlapped_Vector
  end interface

  interface Many_Locus
    module procedure Get_Many_Locus_OV
  end interface

  interface Name
    module procedure Get_Name_Overlapped_Vector
  end interface

  interface One_Locus
    module procedure Get_One_Locus_OV
  end interface

  interface Output
    module procedure Output_Overlapped_Vector
  end interface

  interface Set_Version
    module procedure Set_Version_Overlapped_Vector
  end interface
```

```
interface Version
  module procedure Get_Version_Overlapped_Vector
end interface

! Public type definitions.

public :: Overlapped_Vector_type

type Overlapped_Vector_type

  ! Initialization status.

  type(integer) :: Initialized

  ! The name for this variable (especially useful in a vector of Overlapped
  ! Vectors).

  type(character,name_length) :: Name

  ! Version number which is incremented every time the vector is modified,
  ! or is synced with the version number of a data structure that it
  ! depends on when it is updated.

  type(integer) :: Version

  ! Basic data structures. The Many_Structure corresponds to the structure
  ! of the Distributed Vector that this Overlapped Vector is based on. The
  ! One_Structure corresponds to the way that this Overlapped Vector
  ! has been formed. If this Overlapped Vector were to be combined, it
  ! would result in a Distributed Vector with a One_Structure basis. This
  ! Overlapped Vector can be thought of as a "Many of One" relationship
  ! (e.g. Many Faces of Each Cell, or Faces_of_Cells).

  type(Base_Structure_type), pointer :: Many_Structure
  type(Base_Structure_type), pointer :: One_Structure

  ! The number of dimensions that the "vector" has, including the dimension
  ! that is spread over the processors. "Ragged_Right" vectors are signified
  ! by a Dimensionality of -1.

  type(integer) :: Dimensionality

  ! The extents of the dimensions that the "vector" has, including
  ! the dimension that is spread over the processors, which is last.

  type(integer,1) :: Dimensions

  ! The Distributed Vector that this Overlapped Vector is based on. If
  ! this Overlapped Vector is not based on an external Distributed
  ! Vector, an internal Distributed Vector will be constructed and the
  ! Distributed_Vector variable will point to that one instead.

  type(Distributed_Vector_type), pointer :: DV
```

```
    type(Distributed_Vector_type) :: DV_Internal

    ! The Index that is used to modify the Distributed Vector.

    type(Data_Index_type), pointer :: Many_of_One_Index

    ! Off-PE values in the vector, that are stored locally, with a different
    ! length on each PE. Values may have either 1, 2, 3, or 4 dimensions,
    ! or be a ragged right array. The last dimension is always the dimension
    ! that is spread across the processors. Only one of the following
    ! variables will be allocated for a given object.

    type(real,1) :: Overlap_Values1
    type(real,2) :: Overlap_Values2
    type(real,3) :: Overlap_Values3
    type(real,4) :: Overlap_Values4
    ! Needed for Adaptive Angular Refinement.
    ! type(Ragged_Right_Real_type) :: Overlapped_ValuesRR

    ! The index and trace for the distributed axis of the off-PE values.

    type(integer,1) :: Overlap_Index
    type(Trace_type), pointer :: Overlap_Trace

  end type Overlapped_Vector_type

  ! Global class variables.

  type(integer), parameter :: Version_Increment = 16

contains
```

The Overlapped_Vector Class contains the following routines which are listed in separate sections:

end module Caesar_Overlapped_Vector_Class

D.7.1 Initialize_Overlapped_Vector Procedure

The main documentation of the Initialize_Overlapped_Vector Procedure in § 9.7.1 on page 102 contains additional explanation of this code listing.

```
subroutine Initialize_Overlapped_Vector_1 (OV, DV, Many_of_One_Index, &
                                           Name, status)
  ! Use associations.

  use Caesar_Flags_Module, only: initialized_flag

  ! Input variables.

  type(character,*), intent(in), optional :: Name           ! Variable name.
  type(Distributed_Vector_type), intent(in), target :: DV   ! DV for this OV.
  ! Index for Many-One.
  type(Data_Index_type), intent(in), target :: Many_of_One_Index

  ! Output variables.

  ! Overlapped_Vector to be initialized.
  type(Overlapped_Vector_type), intent(inout) :: OV
  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(Status_type), dimension(1) :: allocate_status  ! Allocation Status.
  type(Status_type) :: consolidated_status            ! Consolidated Status.
  type(integer) :: i, j, k                            ! Loop counters.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(DV),5)                    ! Distributed Vector is valid.
  VERIFY(Valid_State(Many_of_One_Index),5)     ! Index is valid.
  ! Structures are the same.
  VERIFY(ASSOCIATED(Many_of_One_Index%Many_Structure, DV%Structure),5)

  ! Set up pointers.

  OV%One_Structure => Many_of_One_Index%One_Structure
  OV%Many_Structure => DV%Structure
  OV%Dimensions => DV%Dimensions
  OV%DV => DV
  OV%Many_of_One_Index => Many_of_One_Index
  OV%Overlap_Index => Many_of_One_Index%Off_PE_Index
  OV%Overlap_Trace => Many_of_One_Index%Off_PE_Trace

  ! Set up internals.

  if (PRESENT(Name)) OV%Name = Name
  OV%Dimensionality = DV%Dimensionality
```

```
OV%Version = DV%Version

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)

! Set up the Overlapped Values.

select case (OV%Dimensionality)
case (1)
  call Initialize (OV%Overlap_Values1, Many_of_One_Index%NOff_PE, &
                   allocate_status(1))
  call Gather (OV%Overlap_Values1, DV%Values1, Trace=OV%Overlap_Trace)
case (2)
  call Initialize (OV%Overlap_Values2, OV%Dimensions(1), &
                   Many_of_One_Index%NOff_PE, &
                   allocate_status(1))
  do i = 1, OV%Dimensions(1)
    call Gather (OV%Overlap_Values2(i,:), DV%Values2(i,:), &
                 Trace=OV%Overlap_Trace)
  end do
case (3)
  call Initialize (OV%Overlap_Values3, OV%Dimensions(1), &
                   OV%Dimensions(2), Many_of_One_Index%NOff_PE, &
                   allocate_status(1))
  do i = 1, OV%Dimensions(1)
    do j = 1, OV%Dimensions(2)
      call Gather (OV%Overlap_Values3(i,j,:), DV%Values3(i,j,:), &
                   Trace=OV%Overlap_Trace)
    end do
  end do
case (4)
  call Initialize (OV%Overlap_Values4, OV%Dimensions(1), &
                   OV%Dimensions(2), OV%Dimensions(3), &
                   Many_of_One_Index%NOff_PE, &
                   allocate_status(1))
  do i = 1, OV%Dimensions(1)
    do j = 1, OV%Dimensions(2)
      do k = 1, OV%Dimensions(3)
        call Gather (OV%Overlap_Values4(i,j,k,:), DV%Values4(i,j,k,:), &
                     Trace=OV%Overlap_Trace)
      end do
    end do
  end do
!case (-1)
!  call Initialize (OV%Overlap_ValuesRR, Many_of_One_Index%NOff_PE, &
!                   allocate_status)
end select

! Consolidate and handle status.

consolidated_status = allocate_status
if (PRESENT(status)) then
```

```
    WARN_IF(Error(consolidated_status), 5)
    status = consolidated_status
  else
    VERIFY(Normal(consolidated_status), 5)
  end if
  call Finalize (consolidated_status)
  call Finalize (allocate_status)

  ! Set initialization flag.

  OV%Initialized = initialized_flag

  ! Verify guarantees.

  VERIFY(Valid_State(OV),5)  ! OV is now valid.

  return
end subroutine Initialize_Overlapped_Vector_1

subroutine Initialize_Overlapped_Vector_2 (OV, Many_of_One_Index, &
                                           Dimensionality, Name, status, &
                                           dim1, dim2, dim3)

  ! Input variables.

  type(character,*), intent(in), optional :: Name           ! Variable name.
  type(Data_Index_type), intent(in) :: Many_of_One_Index ! Index for Many-1.
  type(integer), intent(in) :: Dimensionality   ! Dimensionality for this OV.
  type(integer), intent(in), optional :: dim1, dim2, dim3   ! Dimensions.

  ! Output variables.

  ! Overlapped_Vector to be initialized.
  type(Overlapped_Vector_type), intent(out) :: OV
  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(Status_type), dimension(2) :: allocate_status ! Allocation Status.
  type(Status_type) :: consolidated_status           ! Consolidated Status.
  ! Pass-through variables.
  type(character,name_length) :: Name_Pass
  type(integer) :: dim1_Pass, dim2_Pass, dim3_Pass

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Many_of_One_Index),5)      ! Index is valid.
  VERIFY(Dimensionality .InInterval. (/1,4/),5) ! Dimensionality is in range.
  VERIFY(PRESENT(dim1) .or. Dimensionality == 1,5) ! Proper dimensions exist.
  VERIFY(PRESENT(dim2) .or. Dimensionality <= 2,5) ! Proper dimensions exist.
  VERIFY(PRESENT(dim3) .or. Dimensionality <= 3,5) ! Proper dimensions exist.
```

```
! Make "Pass" versions of the optional inputs.

if (PRESENT(Name)) then
  Name_Pass = Name
else
  Name_Pass = ''
end if
if (PRESENT(dim1)) then
  dim1_Pass = dim1
else
  dim1_Pass = 0
end if
if (PRESENT(dim2)) then
  dim2_Pass = dim2
else
  dim2_Pass = 0
end if
if (PRESENT(dim3)) then
  dim3_Pass = dim3
else
  dim3_Pass = 0
end if

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)

! Initialize the internal DV.

call Initialize (OV%DV_Internal, Many_of_One_Index%Many_Structure, &
                 Dimensionality, Name_Pass, allocate_status(1), &
                 dim1_Pass, dim2_Pass, dim3_Pass)
!OV%DV_Internal = (/ 0.d0 /)

! Use other OV initialization procedure.

call Initialize_Overlapped_Vector_1 (OV, OV%DV_Internal, &
                                     Many_of_One_Index, Name_Pass, &
                                     allocate_status(2))

! Consolidate and handle status.

consolidated_status = allocate_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
  VERIFY(Normal(consolidated_status), 5)
end if
call Finalize (consolidated_status)
call Finalize (allocate_status)

! Verify guarantees.
```

```
    VERIFY(Valid_State(OV),5)  ! OV is now valid.

    return
  end subroutine Initialize_Overlapped_Vector_2
```

D.7.2 Finalize_Overlapped_Vector Procedure

The main documentation of the Finalize_Overlapped_Vector Procedure in § 9.7.2 on page 102 contains additional explanation of this code listing.

```
  subroutine Finalize_Overlapped_Vector (OV, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Overlapped_Vector to be finalized.
    type(Overlapped_Vector_type), intent(inout) :: OV

    ! Output variable.

    type(Status_type), intent(out), optional :: status   ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(5) :: deallocate_status ! Deallocation Status.
    type(Status_type) :: consolidated_status             ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(OV),7) ! OV is valid.

    ! Unset initialization flag.

    OV%Initialized = uninitialized_flag

    ! Set deallocation status.

    call Initialize (deallocate_status)
    call Initialize (consolidated_status)

    ! Finalize internals.

    NULLIFY(OV%One_Structure)
    NULLIFY(OV%Many_Structure)
    NULLIFY(OV%Dimensions)
    NULLIFY(OV%DV)
```

```
    NULLIFY(OV%Many_of_One_Index)
    NULLIFY(OV%Overlap_Index)
    NULLIFY(OV%Overlap_Trace)

    select case (OV%Dimensionality)
    case (1)
      call Finalize (OV%Overlap_Values1, deallocate_status(1))
    case (2)
      call Finalize (OV%Overlap_Values2, deallocate_status(1))
    case (3)
      call Finalize (OV%Overlap_Values3, deallocate_status(1))
    case (4)
      call Finalize (OV%Overlap_Values4, deallocate_status(1))
    !case (-1)
    !  call Finalize (OV%Overlap_ValuesRR, deallocate_status(1))
    end select
    call Finalize (OV%Dimensionality, deallocate_status(2))
    call Finalize (OV%Name,           deallocate_status(3))
    call Finalize (OV%Version,        deallocate_status(4))
    if (Initialized(OV%DV_Internal)) then
      call Finalize (OV%DV_Internal,  deallocate_status(5))
    end if

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(OV),7) ! OV is not valid.

    return
  end subroutine Finalize_Overlapped_Vector
```

D.7.3 Valid_State_Overlapped_Vector Procedure

The main documentation of the Valid_State_Overlapped_Vector Procedure in § 9.7.3 on page 103 contains additional explanation of this code listing.

```
  function Valid_State_Overlapped_Vector (OV) result(Valid)

    ! Input variables.

    ! Variable to be checked.
```

```
type(Overlapped_Vector_type), intent(in) :: OV

! Output variables.

type(logical) :: Valid       ! Logical state.

! Internal variables.

type(integer) :: NSlice      ! Number of Values in a "slice" of the OV.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Start out true.

Valid = .true.

! Check for association of pointered internals.

Valid = Valid .and. ASSOCIATED(OV%One_Structure)
Valid = Valid .and. ASSOCIATED(OV%Many_Structure)
Valid = Valid .and. ASSOCIATED(OV%Dimensions)
Valid = Valid .and. ASSOCIATED(OV%DV)
Valid = Valid .and. ASSOCIATED(OV%Many_of_One_Index)
Valid = Valid .and. ASSOCIATED(OV%Overlap_Index)
Valid = Valid .and. ASSOCIATED(OV%Overlap_Trace)
if (.not.Valid) return

! Check for validity of internals.

Valid = Valid .and. Initialized(OV)
Valid = Valid .and. Valid_State(OV%Dimensionality)
Valid = Valid .and. Valid_State(OV%Dimensions)
Valid = Valid .and. Valid_State(OV%DV)
Valid = Valid .and. Valid_State(OV%Many_Structure)
Valid = Valid .and. Valid_State(OV%Many_of_One_Index)
Valid = Valid .and. Valid_State(OV%Name)
Valid = Valid .and. Valid_State(OV%One_Structure)
Valid = Valid .and. Valid_State(OV%Overlap_Index)
Valid = Valid .and. Valid_State(OV%Overlap_Trace)
select case (OV%Dimensionality)
case (1)
  Valid = Valid .and. Valid_State(OV%Overlap_Values1)
case (2)
  Valid = Valid .and. Valid_State(OV%Overlap_Values2)
case (3)
  Valid = Valid .and. Valid_State(OV%Overlap_Values3)
case (4)
  Valid = Valid .and. Valid_State(OV%Overlap_Values4)
!case (-1)
!  Valid = Valid .and. Valid_State(OV%Overlap_ValuesRR)
end select
Valid = Valid .and. Valid_State(OV%Version)
if (.not.Valid) return
```

```
    ! Checks on the validity of OV.

    NSlice = PRODUCT(OV%DV%Dimensions(1:OV%Dimensionality-1))
    select case (OV%Dimensionality)
    case (1)
      Valid = Valid .and. &
              OV%Many_of_One_Index%NOff_PE * NSlice == SIZE(OV%Overlap_Values1)
    case (2)
      Valid = Valid .and. &
              OV%Many_of_One_Index%NOff_PE * NSlice == SIZE(OV%Overlap_Values2)
    case (3)
      Valid = Valid .and. &
              OV%Many_of_One_Index%NOff_PE * NSlice == SIZE(OV%Overlap_Values3)
    case (4)
      Valid = Valid .and. &
              OV%Many_of_One_Index%NOff_PE * NSlice == SIZE(OV%Overlap_Values4)
    !case (-1)
    !  Valid = Valid .and. OV%NValues_PE == SIZE(OV%Overlap_ValuesRR)
    end select

    return
  end function Valid_State_Overlapped_Vector
```

D.7.4 Initialized_Overlapped_Vector Procedure

The main documentation of the Initialized_Overlapped_Vector Procedure in § 9.7.4 on page 103 contains additional explanation of this code listing.

```
  function Initialized_Overlapped_Vector (OV) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! Overlapped_Vector to be checked.
    type(Overlapped_Vector_type), intent(in) :: OV

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = OV%Initialized == initialized_flag

    ! Verify guarantees - none.
```

```
  return
end function Initialized_Overlapped_Vector
```

D.7.5 Collect_and_Combine_DV_from_OV Procedure

The main documentation of the Collect_and_Combine_DV_from_OV Procedure in § 9.7.5 on page 104 contains additional explanation of this code listing.

```
define([COLLECT_AND_COMBINE_ROUTINE],[
  pushdef([OP], [$1])
  pushdef([Collect_and_OP_DV_from_OV], expand(Collect_and_$1_DV_from_OV))

  subroutine Collect_and_OP_DV_from_OV (DV, OV)

    ! Input variable.

    type(Overlapped_Vector_type), intent(in) :: OV ! Variable to be combined.

    ! Input/Output variable.

    type(Distributed_Vector_type), intent(inout) :: DV ! Resultant DV.

    ! Internal variables.

    type(integer) :: i, j, k, m, o  ! Loop counters.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(OV),5)                           ! OV is valid.
    VERIFY(Valid_State(DV),5)                           ! DV is valid.
    VERIFY(OV%Dimensionality == DV%Dimensionality,5)    ! Same dimensionality.
    VERIFY(OV%Dimensions(1:OV%Dimensionality-1) == dnl
           DV%Dimensions(1:DV%Dimensionality-1),5)      ! Same dimensions.
    VERIFY(ASSOCIATED(OV%One_Structure,DV%Structure),5) ! Same one-structure.

    ! Collect and combine the values. There are different versions based
    ! on the dimensionality of the Index and on the dimensionality of the
    ! "Vector" itself.

    select case (OV%Many_of_One_Index%Dimensionality)

    ! Vector Index. Shape of DV%Values must be:
    !
    !   DV%Values ( [dim1, [dim2, [dim3, ]]] One_Axis )
    !
    ! There is no Many_Axis to be combined -- no combination operation
    ! is used.

    case (1)
```

```
! Switch on the dimensionality of the data itself.

select case (OV%Dimensionality)

case (1)

  DV%Values1 = OPSTART
  where (OV%Many_of_One_Index%Index1 > 0)
    DV%Values1(:) = OV%DV%Values1(OV%Many_of_One_Index%Index1)
  end where
  where (OV%Many_of_One_Index%Index1 < 0)
    DV%Values1(:) = OV%Overlap_Values1(-OV%Many_of_One_Index%Index1)
  end where

case (2)

  DV%Values2 = OPSTART
  do i = 1, OV%Dimensions(1)
    where (OV%Many_of_One_Index%Index1 > 0)
      DV%Values2(i,:) = OV%DV%Values2(i, OV%Many_of_One_Index%Index1)
    end where
    where (OV%Many_of_One_Index%Index1 < 0)
      DV%Values2(i,:) = &
        OV%Overlap_Values2(i, -OV%Many_of_One_Index%Index1)
    end where
  end do

case (3)

  DV%Values3 = OPSTART
  do i = 1, OV%Dimensions(1)
    do j = 1, OV%Dimensions(2)
      where (OV%Many_of_One_Index%Index1 > 0)
        DV%Values3(i,j,:) = &
          OV%DV%Values3(i, j, OV%Many_of_One_Index%Index1)
      end where
      where (OV%Many_of_One_Index%Index1 < 0)
        DV%Values3(i,j,:) = &
          OV%Overlap_Values3(i, j, -OV%Many_of_One_Index%Index1)
      end where
    end do
  end do

case (4)

  DV%Values4 = OPSTART
  do i = 1, OV%Dimensions(1)
    do j = 1, OV%Dimensions(2)
      do k = 1, OV%Dimensions(3)
        where (OV%Many_of_One_Index%Index1 > 0)
          DV%Values4(i,j,k,:) = &
            OV%DV%Values4(i, j, k, OV%Many_of_One_Index%Index1)
        end where
```

```
          where (OV%Many_of_One_Index%Index1 < 0)
            DV%Values4(i,j,k,:) = &
              OV%Overlap_Values4(i, j, k, -OV%Many_of_One_Index%Index1)
          end where
        end do
      end do
    end do

  end select

! Array Index. Shape of DV%Values must be:
!
!     DV%Values ( [dim1, [dim2, [dim3, ]]] One_Axis )
!
! The Many_Axis has been combined.

case (2)

  ! Switch on the dimensionality of the data itself.

  select case (OV%Dimensionality)

  case (1)

    DV%Values1 = OPSTART
    do o = 1, SIZE(OV%Many_of_One_Index%Index2, 1)
      do m = 1, SIZE(OV%Many_of_One_Index%Index2, 2)
        if (OV%Many_of_One_Index%Index2(o,m) > 0) then
          DV%Values1(o) = OPERATION(DV%Values1(o), dnl
            OV%DV%Values1(OV%Many_of_One_Index%Index2(o,m)))
        else if (OV%Many_of_One_Index%Index2(o,m) < 0) then
          DV%Values1(o) = OPERATION(DV%Values1(o), dnl
            OV%Overlap_Values1(-OV%Many_of_One_Index%Index2(o,m)))
        end if
      end do
      ifelse(OP, [Average], [
        DV%Values1(o) = DV%Values1(o) / &
          changetype(real, COUNT(OV%Many_of_One_Index%Index2(o,:) /= 0))
      ])
    end do

  case (2)

    DV%Values2 = OPSTART
    do i = 1, OV%Dimensions(1)
      do o = 1, SIZE(OV%Many_of_One_Index%Index2, 1)
        do m = 1, SIZE(OV%Many_of_One_Index%Index2, 2)
          if (OV%Many_of_One_Index%Index2(o,m) > 0) then
            DV%Values2(i,o) = OPERATION(DV%Values2(i,o), dnl
              OV%DV%Values2(i, OV%Many_of_One_Index%Index2(o,m)))
          else if (OV%Many_of_One_Index%Index2(o,m) < 0) then
            DV%Values2(i,o) = OPERATION(DV%Values2(i,o), dnl
              OV%Overlap_Values2(i, -OV%Many_of_One_Index%Index2(o,m)))
          end if
```

```
      end do
      ifelse(OP, [Average], [
        DV%Values2(i,o) = DV%Values2(i,o) / &
          changetype(real, COUNT(OV%Many_of_One_Index%Index2(o,:) /= 0))
      ])
    end do
  end do

case (3)

  DV%Values3 = OPSTART
  do i = 1, OV%Dimensions(1)
    do j = 1, OV%Dimensions(2)
      do o = 1, SIZE(OV%Many_of_One_Index%Index2, 1)
        do m = 1, SIZE(OV%Many_of_One_Index%Index2, 2)
          if (OV%Many_of_One_Index%Index2(o,m) > 0) then
            DV%Values3(i,j,o) = OPERATION(DV%Values3(i,j,o), dnl
              OV%DV%Values3(i, j, OV%Many_of_One_Index%Index2(o,m)))
          else if (OV%Many_of_One_Index%Index2(o,m) < 0) then
            DV%Values3(i,j,o) = OPERATION(DV%Values3(i,j,o), dnl
              OV%Overlap_Values3 dnl
              (i, j, -OV%Many_of_One_Index%Index2(o,m)))
          end if
        end do
        ifelse(OP, [Average], [
          DV%Values3(i,j,o) = DV%Values3(i,j,o) / &
            changetype(real, dnl
            COUNT(OV%Many_of_One_Index%Index2(o,:) /= 0))
        ])
      end do
    end do
  end do

case (4)

  DV%Values4 = OPSTART
  do i = 1, OV%Dimensions(1)
    do j = 1, OV%Dimensions(2)
      do k = 1, OV%Dimensions(3)
        do o = 1, SIZE(OV%Many_of_One_Index%Index2, 1)
          do m = 1, SIZE(OV%Many_of_One_Index%Index2, 2)
            if (OV%Many_of_One_Index%Index2(o,m) > 0) then
              DV%Values4(i,j,k,o) = OPERATION(DV%Values4(i,j,k,o), dnl
                OV%DV%Values4 dnl
                (i, j, k, OV%Many_of_One_Index%Index2(o,m)))
            else if (OV%Many_of_One_Index%Index2(o,m) < 0) then
              DV%Values4(i,j,k,o) = OPERATION(DV%Values4(i,j,k,o), dnl
                OV%Overlap_Values4 dnl
                (i, j, k, -OV%Many_of_One_Index%Index2(o,m)))
            end if
          end do
          ifelse(OP, [Average], [
            DV%Values4(i,j,k,o) = DV%Values4(i,j,k,o) / &
              changetype(real, dnl
```

```
                    COUNT(OV%Many_of_One_Index%Index2(o,:) /= 0))
                ])
              end do
            end do
          end do
        end do

      end select

    end select

    ! Set version number.

    DV = Version(OV)

    ! Verify guarantees.

    VERIFY(Valid_State(OV),5)    ! OV is still valid.
    VERIFY(Valid_State(DV),5)    ! DV is valid.

    return
  end subroutine Collect_and_OP_DV_from_OV

  popdef([OP])
  popdef([OPERATION])
  popdef([OPSTART])
  popdef([Collect_and_OP_DV_from_OV])
])

! Add "Conserve" later if needed.

fortext([Op],[Average SUM MAX MIN],[
  ifelse(
    Op, [MAX], [
      pushdef([OPERATION], [Op[]($1, &
      $2)])
      pushdef([OPSTART],[-HUGE(zero)])
    ], Op, [MIN], [
      pushdef([OPERATION], [Op[]($1, &
      $2)])
      pushdef([OPSTART],[HUGE(zero)])
    ], [
      pushdef([OPERATION], [$1 + &
      $2])
      pushdef([OPSTART],[zero])
    ]
  )
  COLLECT_AND_COMBINE_ROUTINE(Op)
])
```

D.7.6 Gather_OV_from_DV Procedure

The main documentation of the Gather_OV_from_DV Procedure in § 9.7.6 on page 104 contains additional explanation of this code listing.

```
subroutine Gather_OV_from_DV (OV, DV)

  ! Input variable.

  type(Distributed_Vector_type), intent(in), target :: DV ! DV to be gathered.

  ! Input/Output variable.

  type(Overlapped_Vector_type), intent(inout) :: OV ! Gathered vector.

  ! Internal variables.

  type(integer) :: i, j, k  ! Loop counters.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(OV),5)                        ! OV is valid.
  VERIFY(Valid_State(DV),5)                        ! DV is valid.
  VERIFY(ASSOCIATED(DV%Structure,OV%Many_Structure),5) ! DV, OV ->same struct
  VERIFY(DV%Dimensionality == OV%Dimensionality,5) ! DV, OV have same Dims.

  ! Set the DV values of the OV if necessary.

  if (.not. ASSOCIATED(OV%DV,DV)) then
    select case (OV%Dimensionality)
    case (1)
      OV%DV%Values1 = DV%Values1
    case (2)
      OV%DV%Values2 = DV%Values2
    case (3)
      OV%DV%Values3 = DV%Values3
    case (4)
      OV%DV%Values4 = DV%Values4
    !case (-1)
    !  OV%DV%ValuesRR = DV%ValuesRR
    end select
  end if

  ! Gather the off PE values if they are out-of-date. This could be done
  ! faster for multiple dimensions by packing everything into a single
  ! vector, gathering, and then unpacking. Maybe in a later version.

  if (Version(OV) /= Version(DV)) then
    select case (OV%Dimensionality)
    case (1)
      call Gather (OV%Overlap_Values1, DV%Values1, Trace=OV%Overlap_Trace)
```

```
      case (2)
        do i = 1, SIZE(OV%Overlap_Values2,1)
          call Gather (OV%Overlap_Values2(i,:), DV%Values2(i,:), &
                       Trace=OV%Overlap_Trace)
        end do
      case (3)
        do i = 1, SIZE(OV%Overlap_Values3,1)
          do j = 1, SIZE(OV%Overlap_Values3,2)
            call Gather (OV%Overlap_Values3(i,j,:), DV%Values3(i,j,:), &
                         Trace=OV%Overlap_Trace)
          end do
        end do
      case (4)
        do i = 1, SIZE(OV%Overlap_Values4,1)
          do j = 1, SIZE(OV%Overlap_Values4,2)
            do k = 1, SIZE(OV%Overlap_Values4,3)
              call Gather (OV%Overlap_Values4(i,j,k,:), DV%Values4(i,j,k,:), &
                           Trace=OV%Overlap_Trace)
            end do
          end do
        end do
      !case (-1)
      !  call Gather (OV%Overlap_ValuesRR, DV%ValuesRR, &
      !               Trace=OV%Overlap_Trace)
      end select
    end if

    ! Set version.

    OV = Version(DV)

    ! Verify guarantees.

    VERIFY(Valid_State(OV),5)  ! OV is valid.

    return
  end subroutine Gather_OV_from_DV
```

D.7.7 Get_Locus_Overlapped_Vector Procedure

The main documentation of the Get_Locus_Overlapped_Vector Procedure in § 9.7.7 on page 105 contains additional explanation of this code listing.

```
  define([LOCUS_ROUTINE],[
    pushdef([Get_MANYORONE_Locus_OV], expand(Get_$1_Locus_OV))

    function Get_MANYORONE_Locus_OV (OV) result(Locus_OV)

      ! Input variable.

      type(Overlapped_Vector_type), intent(in) :: OV   ! Variable to be queried.
```

```
      ! Output variable.

      type(character,name_length) :: Locus_OV          ! Locus of OV.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(OV),5)      ! OV is valid.

      ! Set the value.

      Locus_OV = Locus(OV%$1_Structure)

      ! Verify guarantees - none.

      return
    end function Get_MANYORONE_Locus_OV

    popdef([Get_MANYORONE_Locus_OV])
  ])

  LOCUS_ROUTINE(Many)
  LOCUS_ROUTINE(One)
```

D.7.8 Get_Name_Overlapped_Vector Procedure

The main documentation of the Get_Name_Overlapped_Vector Procedure in § 9.7.8 on page 105 contains additional explanation of this code listing.

```
  function Get_Name_Overlapped_Vector (OV) result(Name)

    ! Input variable.

    type(Overlapped_Vector_type), intent(in) :: OV  ! Variable to be queried.

    ! Output variable.

    type(character,name_length) :: Name             ! Name of OV.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(OV),5)      ! OV is valid.

    ! Set the value.

    Name = OV%Name

    ! Verify guarantees - none.
```

```
    return
  end function Get_Name_Overlapped_Vector
```

D.7.9 Get_Values_Overlapped_Vector Procedure

The main documentation of the Get_Values_Overlapped_Vector Procedure in § 9.7.9 on page 105 contains additional explanation of this code listing.

```
  define([GET_VALUES_ROUTINE],[
    pushdef([DIM], [$1])
    pushdef([DIMS],
      [ifelse(
        [$1], [1],
          [],
        [forloop([i],2,$1,[:,])])])
    pushdef([Get_Values_Overlapped_Vector_DIM],
      expand(Get_Values_Overlapped_Vector_DIM))

    subroutine Get_Values_Overlapped_Vector_DIM (Values, OV)

      ! Input variable.

      type(Overlapped_Vector_type), intent(in) :: OV ! Variable to be queried.

      ! Input/Output variable.

      type(real,DIM,np), intent(inout) :: Values     ! Values bare naked array.

      ! Internal variables.

      ifelse(DIM, [1], [
      ], DIM, [2], [
        type(integer) :: i, m       ! Loop counters.
      ], DIM, [3], [
        type(integer) :: i, j, m    ! Loop counters.
      ], DIM, [4], [
        type(integer) :: i, j, k, m ! Loop counters.
      ], DIM, [5], [
        type(integer) :: i, j, k, m ! Loop counters.
      ])
      ifelse(m4_eval(DEBUG_LEVEL >= 5), 1, [
        type(integer) :: IndexDim ! OV%Many_of_One_Index%Dimensionality, used
                                  ! in VERIFY commands below.
        type(integer) :: Many_Axis_Size ! Number of Manys for each One.
      ])

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verification setup -- not done unless VERIFYs are turned on. The
      ! activation level used here should correspond to the one used in the
      ! VERIFY commands below. Also, see similar construct in declarations
      ! above.
```

```
ifelse(m4_eval(DEBUG_LEVEL >= 5), 1, [

  ! Define shorter form of OV%Many_of_One_Index%Dimensionality to
  ! avoid line length problems.
  IndexDim = OV%Many_of_One_Index%Dimensionality

  ! Calculate Many Axis size.
  if (IndexDim == 2) then
    Many_Axis_Size = SIZE(OV%Many_of_One_Index%Index2, 2)
  else if (IndexDim == 1) then
    Many_Axis_Size = 1
  else
    ! Shouldn't be triggered. Will add something for RR here later.
    VERIFY(.false.,0)
  end if

])

! Verify requirements.

VERIFY(Valid_State(OV),5)                      ! OV is valid.
VERIFY(Valid_State_NP(Values),5)               ! Values is valid.
! OV has been set up for this call.
VERIFY(DIM .InInterval. (/ OV%Dimensionality, OV%Dimensionality+1 /),5)
! Values shape checks:
!
!   Shape is
!     Values ( [dim1, [dim2, [dim3, ]]] One_Axis [, Many_Axis] )
!
!   First axes are OV%Dimensions.
VERIFY(SIZE(Values,MIN(1,DIM)) == OV%Dimensions(1) .or. dnl
       OV%Dimensionality == 1,5)
VERIFY(SIZE(Values,MIN(2,DIM)) == OV%Dimensions(2) .or. dnl
       OV%Dimensionality <= 2,5)
VERIFY(SIZE(Values,MIN(3,DIM)) == OV%Dimensions(3) .or. dnl
       OV%Dimensionality <= 3,5)
!   Penultimate axis is One_Structure axis if Index is two-dimensional.
VERIFY(SIZE(Values,MAX(1,DIM-1)) == Length_PE(OV%One_Structure) .or. dnl
       IndexDim /= 2, 5)
!   Last axis is One_Structure axis if Index is a vector index.
VERIFY(SIZE(Values,DIM) == Length_PE(OV%One_Structure) .or.          dnl
       IndexDim /= 1, 5)
!   Last axis is Many_Structure axis if Index is two-dimensional.
VERIFY(SIZE(Values,DIM) == Many_Axis_Size .or.     dnl
       IndexDim /= 2, 5)

! Collect and set the values. There are different versions based on the
! dimensionality of the Index and on the dimensionality of the "Vector"
! itself. Note that there will only be two versions (for the Index
! dimensionality) for each routine in the m4-preprocessed file.

Values = zero
select case (OV%Many_of_One_Index%Dimensionality)
```

```
! Vector Index. Shape of Values must be:
!
!     Values ( [dim1, [dim2, [dim3, ]]] One_Axis )

case (1)

  ifelse(DIM, [1], [

    where (OV%Many_of_One_Index%Index1 > 0)
      Values(:) = &
        OV%DV%Values1(OV%Many_of_One_Index%Index1)
    end where
    where (OV%Many_of_One_Index%Index1 < 0)
      Values(:) = OV%Overlap_Values1(-OV%Many_of_One_Index%Index1)
    end where

  ], DIM, [2], [

    do i = 1, OV%Dimensions(1)
      where (OV%Many_of_One_Index%Index1 > 0)
        Values(i,:) = &
          OV%DV%Values2(i, OV%Many_of_One_Index%Index1)
      end where
      where (OV%Many_of_One_Index%Index1 < 0)
        Values(i,:) = OV%Overlap_Values2(i, -OV%Many_of_One_Index%Index1)
      end where
    end do

  ], DIM, [3], [

    do i = 1, OV%Dimensions(1)
      do j = 1, OV%Dimensions(2)
        where (OV%Many_of_One_Index%Index1 > 0)
          Values(i,j,:) = &
            OV%DV%Values3(i, j, OV%Many_of_One_Index%Index1)
        end where
        where (OV%Many_of_One_Index%Index1 < 0)
          Values(i,j,:) = &
            OV%Overlap_Values3(i, j, -OV%Many_of_One_Index%Index1)
        end where
      end do
    end do

  ], DIM, [4], [

    do i = 1, OV%Dimensions(1)
      do j = 1, OV%Dimensions(2)
        do k = 1, OV%Dimensions(3)
          where (OV%Many_of_One_Index%Index1 > 0)
            Values(i,j,k,:) = &
              OV%DV%Values4(i, j, k, OV%Many_of_One_Index%Index1)
          end where
          where (OV%Many_of_One_Index%Index1 < 0)
```

```
            Values(i,j,k,:) = &
              OV%Overlap_Values4(i, j, k, -OV%Many_of_One_Index%Index1)
          end where
        end do
      end do
    end do

  ], DIM, [5], [

    ! This combination shouldn't be triggered.
    VERIFY(.false., 0)

  ])

! Array Index. Shape of Values must be:
!
!     Values ( [dim1, [dim2, [dim3, ]]] One_Axis, Many_Axis )

case (2)

  ifelse(DIM, [1], [

    ! This combination shouldn't be triggered.
    VERIFY(.false., 0)

  ], DIM, [2], [

    do m = 1, SIZE(OV%Many_of_One_Index%Index2, 2)
      where (OV%Many_of_One_Index%Index2(:,m) > 0)
        Values(:,m) = &
          OV%DV%Values1(OV%Many_of_One_Index%Index2(:,m))
      end where
      where (OV%Many_of_One_Index%Index2(:,m) < 0)
        Values(:,m) = &
          OV%Overlap_Values1(-OV%Many_of_One_Index%Index2(:,m))
      end where
    end do

  ], DIM, [3], [

    do i = 1, OV%Dimensions(1)
      do m = 1, SIZE(OV%Many_of_One_Index%Index2, 2)
        where (OV%Many_of_One_Index%Index2(:,m) > 0)
          Values(i,:,m) = &
            OV%DV%Values2(i, OV%Many_of_One_Index%Index2(:,m))
        end where
        where (OV%Many_of_One_Index%Index2(:,m) < 0)
          Values(i,:,m) = &
            OV%Overlap_Values2(i, -OV%Many_of_One_Index%Index2(:,m))
        end where
      end do
    end do

  ], DIM, [4], [
```

```
        do i = 1, OV%Dimensions(1)
          do j = 1, OV%Dimensions(2)
            do m = 1, SIZE(OV%Many_of_One_Index%Index2, 2)
              where (OV%Many_of_One_Index%Index2(:,m) > 0)
                Values(i,j,:,m) = &
                  OV%DV%Values3(i, j, OV%Many_of_One_Index%Index2(:,m))
              end where
              where (OV%Many_of_One_Index%Index2(:,m) < 0)
                Values(i,j,:,m) = &
                  OV%Overlap_Values3(i, j, -OV%Many_of_One_Index%Index2(:,m))
              end where
            end do
          end do
        end do

      ], DIM, [5], [

        do i = 1, OV%Dimensions(1)
          do j = 1, OV%Dimensions(2)
            do k = 1, OV%Dimensions(3)
              do m = 1, SIZE(OV%Many_of_One_Index%Index2, 2)
                where (OV%Many_of_One_Index%Index2(:,m) > 0)
                  Values(i,j,k,:,m) = &
                    OV%DV%Values4(i, j, k, OV%Many_of_One_Index%Index2(:,m))
                end where
                where (OV%Many_of_One_Index%Index2(:,m) < 0)
                  Values(i,j,k,:,m) = &
                    OV%Overlap_Values4 &
                    (i, j, k, -OV%Many_of_One_Index%Index2(:,m))
                end where
              end do
            end do
          end do
        end do

      ])

    end select

    ! Verify guarantees.

    VERIFY(Valid_State(OV),5)          ! OV is still valid.
    VERIFY(Valid_State_NP(Values),5)   ! Values is valid.

    return
  end subroutine Get_Values_Overlapped_Vector_DIM

  popdef([DIM])
  popdef([Get_Values_Overlapped_Vector_DIM])
])

forloop([Dim],[1],[5],[
  GET_VALUES_ROUTINE(Dim)
```

```
])
```

D.7.10 Get_Version_Overlapped_Vector Procedure

The main documentation of the Get_Version_Overlapped_Vector Procedure in § 9.7.10 on page 106 contains additional explanation of this code listing.

```
function Get_Version_Overlapped_Vector (OV) result(Version)

  ! Input variable.

  type(Overlapped_Vector_type), intent(in) :: OV   ! Variable to be queried.

  ! Output variable.

  type(integer) :: Version                         ! Version number.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(OV),5)      ! OV is valid.

  ! Get the value.

  Version = OV%Version

  ! Verify guarantees - none.

  return
end function Get_Version_Overlapped_Vector
```

D.7.11 Output_Overlapped_Vector Procedure

The main documentation of the Output_Overlapped_Vector Procedure in § 9.7.11 on page 106 contains additional explanation of this code listing.

```
subroutine Output_Overlapped_Vector (OV, Many_First, Many_Last, One_First, &
                                     One_Last, Unit)

  ! Input variables.

  type(Overlapped_Vector_type), intent(in) :: OV    ! Variable to be output.
  type(integer), intent(in), optional :: Many_First ! Extents of many value
  type(integer), intent(in), optional :: Many_Last  !   data to be output.
  type(integer), intent(in), optional :: One_First  ! Extents of one value
  type(integer), intent(in), optional :: One_Last   !   data to be output.
  type(integer), intent(in), optional :: Unit       ! Output unit.
```

```
! Internal variables.

type(integer) :: Buffer_Loc                    ! Buffer location.
type(integer) :: Buffer_Size                   ! Output buffer size.
type(integer) :: Buffer_Skip                   ! Buffer increment.
type(integer) :: i                             ! Loop counter.
type(integer) :: A_Many_First                  ! Actual many first value.
type(integer) :: A_Many_Last                   ! Actual many last value.
type(integer) :: A_One_First                   ! Actual one first value.
type(integer) :: A_One_Last                    ! Actual one last value.
type(integer) :: A_Unit                        ! Actual output unit.
type(character,80) :: Name_Name                ! Name of the OV.
type(character,80) :: Output_1                 ! Output buffer.
type(character,80,1) :: Output_Buffer          ! Output buffer vector.
type(integer) :: Version_Number                ! Version of the OV.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(OV),5)      ! OV is valid.

! Set unit number.

if (PRESENT(Unit)) then
  A_Unit = Unit
else
  A_Unit = 6
end if

! These are evaluated on all PEs -- NOT inside an IO PE block -- because
! they contain validity checks on OV and thus require global communication.

Version_Number = Version(OV)
Name_Name = Name(OV)

! Output Identification Info.

if (this_is_IO_PE) then
  write (A_Unit,100) 'Overlapped Vector Information:'
  write (A_Unit,*) ' Name                    = ', TRIM(Name_Name)
  write (A_Unit,*) ' Locus                   = ', &
                 TRIM(Locus(OV%Many_Structure)), ' of ', &
                 TRIM(Locus(OV%One_Structure))
  write (A_Unit,*) ' Initialized             = ', Initialized(OV)
  write (A_Unit,*) ' Version                 = ', Version_Number
  write (A_Unit,*) ' Internal DV             = ', &
                 Initialized(OV%DV_Internal)
  write (A_Unit,*) ' Dimensionality          = ', OV%Dimensionality
end if

! PE-dependent info.

Buffer_Size = MAX(1, (SIZE(OV%Overlap_Index) + 3)/ 4)
```

```
    call Initialize (Output_Buffer, Buffer_Size)

    write (Output_1,102) 'PE:', this_PE, &
                         ', Dimensions    =', OV%Dimensions
    call Parallel_Write (Output_1, A_Unit)

    !write (Output_Buffer(1),102) 'PE:', this_PE, &
    !                             ', NOff_PE =', OV%NOff_PE
    !call Parallel_Write (Output_Buffer(1), A_Unit)
    write (Output_Buffer,104) 'PE:', this_PE, &
                              ', Overlap_Index =', OV%Overlap_Index
    call Parallel_Write (Output_Buffer, A_Unit)

    call Finalize (Output_Buffer)

    ! Set up actual limit values for pass-through.

    if (PRESENT(Many_First)) then
      A_Many_First = Many_First
    else
      A_Many_First = 1
    end if
    if (PRESENT(Many_Last)) then
      A_Many_Last = Many_Last
    else
      A_Many_Last = Length_Total(OV%Many_Structure)
    end if
    if (PRESENT(One_First)) then
      A_One_First = One_First
    else
      A_One_First = 1
    end if
    if (PRESENT(One_Last)) then
      A_One_Last = One_Last
    else
      A_One_Last = Length_Total(OV%One_Structure)
    end if

    ! Output internal structure info.

    call Output (OV%DV, A_Many_First, A_Many_Last, A_Unit, Indent=2)
    call Output (OV%Many_of_One_Index, A_One_First, A_One_Last, A_Unit, &
                 Indent=2)
    !call Output (OV%Trace, A_Unit)
    !call Output (OV%Off_PE_Trace, A_Unit)

    ! Output internal values.

    if (this_is_IO_PE) then
      write (A_Unit,100) '  Internal Overlap Values:'
    end if

    ! Output the values based on the dimensionality.
    ! (In the future, I could limit this with the First and Last limits above,
```

```
! but for now, just output the whole thing.)

select case (OV%Dimensionality)
case (1)
  Buffer_Size = MAX(0, OV%Many_of_One_Index%NOff_PE)
  call Initialize (Output_Buffer, Buffer_Size)
  if (Buffer_Size /= 0) then
    Buffer_Skip = 1
    Buffer_Loc = 1
    do i = 1, OV%Many_of_One_Index%NOff_PE
      write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),103) &
        'PE:', this_PE, ', O_Values1(', i, ') =', &
        OV%Overlap_Values1(i)
      Buffer_Loc = Buffer_Loc + Buffer_Skip
    end do
  end if
case (2)
  Buffer_Size = MAX(0, ((SIZE(OV%Overlap_Values2(:,1)) + 2) / 3) &
                * OV%Many_of_One_Index%NOff_PE)
  call Initialize (Output_Buffer, Buffer_Size)
  if (Buffer_Size /= 0) then
    Buffer_Skip = (SIZE(OV%Overlap_Values2(:,1)) + 2) / 3
    Buffer_Loc = 1
    do i = 1, OV%Many_of_One_Index%NOff_PE
      write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),103) &
        'PE:', this_PE, ', O_Values2(:,', i, ') =', &
        OV%Overlap_Values2(:,i)
      Buffer_Loc = Buffer_Loc + Buffer_Skip
    end do
  end if
case (3)
  Buffer_Size = MAX(0, ((SIZE(OV%Overlap_Values3(:,:,1)) + 2) / 3) &
                * OV%Many_of_One_Index%NOff_PE)
  call Initialize (Output_Buffer, Buffer_Size)
  if (Buffer_Size /= 0) then
    Buffer_Skip = (SIZE(OV%Overlap_Values3(:,:,1)) + 2) / 3
    Buffer_Loc = 1
    do i = 1, OV%Many_of_One_Index%NOff_PE
      write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),103) &
        'PE:', this_PE, ', O_Values3(:,:,', i, ') =', &
        OV%Overlap_Values3(:,:,i)
      Buffer_Loc = Buffer_Loc + Buffer_Skip
    end do
  end if
case (4)
  Buffer_Size = MAX(0, ((SIZE(OV%Overlap_Values4(:,:,:,1)) + 2) / 3) &
                * OV%Many_of_One_Index%NOff_PE)
  call Initialize (Output_Buffer, Buffer_Size)
  if (Buffer_Size /= 0) then
    Buffer_Skip = (SIZE(OV%Overlap_Values4(:,:,:,1)) + 2) / 3
    Buffer_Loc = 1
    do i = 1, OV%Many_of_One_Index%NOff_PE
      write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),103) &
        'PE:', this_PE, ', O_Values4(:,:,:,', i, ') =', &
```

```
          OV%Overlap_Values4(:,:,:,i)
        Buffer_Loc = Buffer_Loc + Buffer_Skip
      end do
    end if
!    case (-1)
!      Buffer_Size = MAX(0, ((SIZE(OV%Overlap_ValuesRR(:,1)) + 2) / 3) &
!                    * OV%Many_of_One_Index%NOff_PE)
!      call Initialize (Output_Buffer, Buffer_Size)
!      if (Buffer_Size /= 0) then
!        Buffer_Skip = (SIZE(OV%Overlap_ValuesRR(:,1)) + 2) / 3
!        Buffer_Loc = 1
!        do i = 1, OV%Many_of_One_Index%NOff_PE
!          write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),103) &
!            'PE:', this_PE, ', O_ValuesRR(:,', i, ') =', &
!            OV%Overlap_ValuesRR(:,i)
!          Buffer_Loc = Buffer_Loc + Buffer_Skip
!        end do
!      end if
    end select
    call Parallel_Write (Output_Buffer, A_Unit)
    call Finalize (Output_Buffer)

    ! Format statements. With these formats, this should work up to
    ! (10^6 - 1) PEs.

100 format (/, a, /)
101 format (2x, a, i11, :, 3(',', i11, :), ',', /, &
            (24x, i11, :, 3(',', i11, :), ','))
102 format (2x, a, i5, a, i11, :, 4(',', i11, :))
103 format (2x, a, i5, a, i11, a, 1p, e13.5e3, :, &
            2(',', e13.5e3, :), ',', /, &
            (38x, e13.5e3, :, 2(',', e13.5e3, :), ','))
104 format (2x, a, i5, a, i11, :, &
            3(',', i11, :), ',', /, &
            (27x, i11, :, 3(',', i11, :), ','))

    ! Verify guarantees - none.

    return
  end subroutine Output_Overlapped_Vector
```

D.7.12 Set_Version_Overlapped_Vector Procedure

The main documentation of the Set_Version_Overlapped_Vector Procedure in § 9.7.12 on page 106 contains additional explanation of this code listing.

```
  subroutine Set_Version_Overlapped_Vector (OV, Version)

    ! Input variable.

    type(integer), intent(in) :: Version                ! Version number.
```

```
    ! Input/Output variable.

    type(Overlapped_Vector_type), intent(inout) :: OV  ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(OV),5)      ! OV is valid.

    ! Set the value.

    OV%Version = Version

    ! Verify guarantees - none.

    return
  end subroutine Set_Version_Overlapped_Vector
```

D.7.13 Overlapped_Vector Class Unit Test Program

This lightly commented program performs a unit test on the Overlapped_Vector Class, which is described in § 9.7 on page 100.

```
program Unit_Test
  use Caesar_Intrinsics_Module
  use Caesar_Base_Structure_Class
  use Caesar_Data_Index_Class
  use Caesar_Assembled_Vector_Class
  use Caesar_Distributed_Vector_Class
  use Caesar_Overlapped_Vector_Class
  use Caesar_Communication_Class
  use Caesar_Numbers_Module, only: one, four
  implicit none

  type(Communication_type) :: Comm
  type(Base_Structure_type) :: Cell_Structure, Node_Structure
  type(integer,2) :: Nodes_of_Cells_Index_Values
  type(Data_Index_type) :: Nodes_of_Cells_Index
  type(Assembled_Vector_type) :: Coordinates_Nodes_AV
  type(Distributed_Vector_type) :: Coordinates_Nodes_DV, Results_Cells_DV
  type(Overlapped_Vector_type) :: Coordinates_Nodes_of_Cells_OV
  type(Status_type) :: status
  type(character,name_length) :: Name_Name
  type(real,2) :: Coordinates, Results_Cells_BNV
  type(real,3) :: Processed_Coordinates
  type(integer) :: NodeSize, Dimensionality, DimSize, &
                   Many_Axis_Length, n, NDimensions, Nodes_per_Cell, &
                   NNodes
  type(logical) :: detailed_output, Success

  ! Initializations.
```

```
call Initialize (Comm)
call Output (Comm)
call Initialize (status)
call Initialize (Name_Name)
Dimensionality = 2
NDimensions = 2
detailed_output = NPEs <= 8

! Set up the Shell Partition Structures.

call Initialize_Shell_Partition (NDimensions, Cell_Structure, &
                                 Node_Structure, Nodes_of_Cells_Index, &
                                 detailed_output)

! Initialize AV, DV and OV Coordinate vectors.

Name_Name = 'Coordinates of Nodes'
call Initialize (Coordinates_Nodes_AV, Node_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)
call Initialize (Coordinates_Nodes_DV, Node_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)
Name_Name = 'Coordinates of Nodes of Cells'
call Initialize (Coordinates_Nodes_of_Cells_OV, Nodes_of_Cells_Index, &
                 Dimensionality, Name_Name, status, NDimensions)
Name_Name = 'Results of Cells'
call Initialize (Results_Cells_DV, Cell_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)

! Set up Coordinates array on IO PE only.

NNodes = Length_Total(Node_Structure)
if (this_is_IO_PE) then
  DimSize = NDimensions
  NodeSize = NNodes
else
  DimSize = 0
  NodeSize = 0
end if
call Initialize (Coordinates, DimSize, NodeSize)
if (this_is_IO_PE) then
  Coordinates(1,:) = (/ ( changetype(real,(n)), n = 1,NNodes ) /)
  Coordinates(2,:) = (/ ( one, n = 1,NNodes ) /)
end if

Name_Name = ''

! Set up Processed Coordinates array on every processor.

Nodes_per_Cell = 2**NDimensions
Many_Axis_Length = Nodes_per_Cell
call Initialize (Processed_Coordinates, NDimensions, &
                 Length_PE(Cell_Structure), Many_Axis_Length)
```

```
! Set up Results_Cells_BNV array on every processor.

Nodes_per_Cell = 2**NDimensions
call Initialize (Results_Cells_BNV, NDimensions, &
                 Length_PE(Cell_Structure))

! Version number check.

Coordinates_Nodes_of_Cells_OV = 123
Success = Version(Coordinates_Nodes_of_Cells_OV) == 123
call Output_Test ('Version number', Success)

! Send Coordinates into Assembled Vector, then Distributed Vector,
! then Overlapped Vector, and access the data.

Coordinates_Nodes_AV = Coordinates
Coordinates_Nodes_DV = Coordinates_Nodes_AV
Coordinates_Nodes_of_Cells_OV = Coordinates_Nodes_DV
Processed_Coordinates = Coordinates_Nodes_of_Cells_OV

! Re-construct the original Nodes_of_Cells index values for comparison.

call Initialize (Nodes_of_Cells_Index_Values, &
                 SIZE(Nodes_of_Cells_Index%Index2,1), &
                 SIZE(Nodes_of_Cells_Index%Index2,2))
Nodes_of_Cells_Index_Values = Nodes_of_Cells_Index

! Check to see if the Processed Coordinates are correct.

Success = Global_ALL(INT(Processed_Coordinates(1,:,:)) == &
          Nodes_of_Cells_Index_Values(:,:))
call Output_Test ('Index', Success)

Success = Global_ALL(Processed_Coordinates(2,:,:) == one)
call Output_Test ('One', Success)

! Combination tests:

!   Average test.

call Collect_and_Average (Results_Cells_DV, Coordinates_Nodes_of_Cells_OV)
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(Results_Cells_BNV(1,:) == &
          changetype(real, SUM(Nodes_of_Cells_Index_Values(:,:),2)) /  &
          changetype(real, Nodes_per_Cell))
call Output_Test ('Average index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Average one', Success)

!   Max test.

call Collect_and_MAX (Results_Cells_DV, Coordinates_Nodes_of_Cells_OV)
Results_Cells_BNV = Results_Cells_DV
```

```
Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          MAXVAL(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Max index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Max one', Success)

!   Min test.

call Collect_and_MIN (Results_Cells_DV, Coordinates_Nodes_of_Cells_OV)
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          MINVAL(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Min index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Min one', Success)

!   Sum test.

Results_Cells_DV = Coordinates_Nodes_of_Cells_OV  ! Sum is the default.
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          SUM(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Sum index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == four)
call Output_Test ('Sum four', Success)

! Output statements.

call Output (Coordinates_Nodes_of_Cells_OV, &
  MAX(1, Length_Total(Node_Structure)/10), &
  MIN(Length_Total(Node_Structure), Length_Total(Node_Structure)/10+50), &
  MAX(1, Length_Total(Cell_Structure)/10), &
  MIN(Length_Total(Cell_Structure), Length_Total(Cell_Structure)/10+50) &
)

! Check state of various objects.

VERIFY(Valid_State(Coordinates_Nodes_AV),0)
VERIFY(Valid_State(Coordinates_Nodes_DV),0)
VERIFY(Valid_State(Coordinates_Nodes_of_Cells_OV),0)
VERIFY(Valid_State(Cell_Structure),0)
VERIFY(Valid_State(Node_Structure),0)
VERIFY(Valid_State(Nodes_of_Cells_Index),0)

! Finalize data structures and communications.

call Finalize (Results_Cells_DV)
call Finalize (Results_Cells_BNV)
call Finalize (Nodes_of_Cells_Index_Values)
call Finalize (Coordinates_Nodes_of_Cells_OV)
call Finalize (Coordinates_Nodes_AV)
```

```
  call Finalize (Coordinates_Nodes_DV)
  call Finalize (Coordinates)
  call Finalize (Processed_Coordinates)
  call Finalize (Nodes_of_Cells_Index)
  call Finalize (Cell_Structure)
  call Finalize (Node_Structure)
  call Finalize (Comm)

end
```

D.8 Collected_Array Class Code Listing

The main documentation of the Collected_Array Class in § 9.8 on page 107 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 01/21/00
! CVS Info:   $Id: collected_array.F90,v 10.9 2008/09/30 16:19:07 hall Exp $

module Caesar_Collected_Array_Class

  ! Global use associations.

  use Caesar_Intrinsics_Module
  use Caesar_Trace_Class
  use Caesar_Communication_Class
  use Caesar_Base_Structure_Class
  use Caesar_Data_Index_Class
  use Caesar_Assembled_Vector_Class
  use Caesar_Distributed_Vector_Class
  use Caesar_Overlapped_Vector_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Assignment (=), Collect, Combine_with_Average, Combine_with_MAX, &
            Combine_with_MIN, Combine_with_SUM, Gather, Gather_and_Collect, &
            Get_Values, Many_Locus, Name, One_Locus, Output, Set_Values, &
            Set_Version, Version

  interface Initialize
```

```
    module procedure Initialize_Collected_Array_1
    module procedure Initialize_Collected_Array_2
  end interface

  interface Finalize
    module procedure Finalize_Collected_Array
  end interface

  interface Valid_State
    module procedure Valid_State_Collected_Array
  end interface

  interface Assignment (=)
    module procedure Collect_CA_from_OV
    module procedure Gather_and_Collect_CA_from_DV
    module procedure Get_Values_Collected_Array_1
    module procedure Get_Values_Collected_Array_2
    module procedure Get_Values_Collected_Array_3
    module procedure Get_Values_Collected_Array_4
    module procedure Get_Values_Collected_Array_5
    module procedure Set_Values_Collected_Array_1
    module procedure Set_Values_Collected_Array_2
    module procedure Set_Values_Collected_Array_3
    module procedure Set_Values_Collected_Array_4
    module procedure Set_Values_Collected_Array_5
    module procedure Set_Version_Collected_Array
    module procedure Combine_with_SUM_DV_from_CA
  end interface

  interface Collect
    module procedure Collect_CA_from_OV
  end interface

  fortext([Op],[Average MAX MIN SUM],[
    interface expand(Combine_with_Op)
      module procedure expand(Combine_with_Op_DV_from_CA)
    end interface
  ])

  interface Gather
    module procedure Gather_and_Collect_CA_from_DV
  end interface

  interface Gather_and_Collect
    module procedure Gather_and_Collect_CA_from_DV
  end interface

  interface Get_Values
    module procedure Get_Values_Collected_Array_1
    module procedure Get_Values_Collected_Array_2
    module procedure Get_Values_Collected_Array_3
    module procedure Get_Values_Collected_Array_4
    module procedure Get_Values_Collected_Array_5
  end interface
```

```
interface Initialized
  module procedure Initialized_Collected_Array
end interface

interface Many_Locus
  module procedure Get_Many_Locus_CA
end interface

interface Name
  module procedure Get_Name_Collected_Array
end interface

interface One_Locus
  module procedure Get_One_Locus_CA
end interface

interface Output
  module procedure Output_Collected_Array
end interface

interface Set_Values
  module procedure Set_Values_Collected_Array_1
  module procedure Set_Values_Collected_Array_2
  module procedure Set_Values_Collected_Array_3
  module procedure Set_Values_Collected_Array_4
  module procedure Set_Values_Collected_Array_5
end interface

interface Set_Version
  module procedure Set_Version_Collected_Array
end interface

interface Version
  module procedure Get_Version_Collected_Array
end interface

! Public type definitions.

public :: Collected_Array_type

type Collected_Array_type

  ! Initialization flag.

  type(integer) :: Initialized

  ! The name for this variable (especially useful in a vector of Collected
  ! Arrays).

  type(character,name_length) :: Name

  ! Version number which is incremented every time the array is modified,
  ! or is synced with the version number of a data structure that it
```

```
! depends on when it is updated.

type(integer) :: Version

! Basic data structures. The Many_Structure corresponds to the structure
! of the Distributed Vector that this Collected Array is based on. The
! One_Structure corresponds to the way that this Collected Array
! has been formed. If this Collected Array were to be combined, it
! would result in a Distributed Vector with a One_Structure basis. This
! Collected Array can be thought of as a "Many of One" relationship
! (e.g. Many Faces of Each Cell, or Faces_of_Cells).

type(Base_Structure_type), pointer :: Many_Structure
type(Base_Structure_type), pointer :: One_Structure

! The number of dimensions that the "array" has, including the dimension
! that is spread over the processors (the One_Axis), but not including
! the Many_Axis, if it is present. "Ragged_Right" vectors are signified
! by a Dimensionality of -1. Note that the Collected Array will have the
! same Dimensionality as the Distributed Vector it is based on and the
! Distributed Vector it can be combined to form, even though in reality
! it may have an extra dimension. The "Actual Dimensionality", which
! potentially includes the Many_Axis, is given by A_Dimensionality.

type(integer) :: Dimensionality
type(integer) :: A_Dimensionality

! The extents of the dimensions that the "array" has, including
! the dimensions for the One_Axis and the Many_Axis (if present).

type(integer,1) :: Dimensions

! The Index that is used to translate between the Distributed Vectors.

type(Data_Index_type), pointer :: Many_of_One_Index

! Values in the array, that are stored locally, with a different
! length on each PE. Values may have either 1, 2, 3, 4, or 5 dimensions,
! or be a ragged right array. The last dimension is the dimension
! that is spread across the processors, if the Many_of_One_Index is a
! vector index. Otherwise, the penultimate axis will be spread across the
! processors. The form of the Values array is:
!
!   Values( [dim1, [dim2, [dim3,]]] One_Axis [, Many_Axis] )
!
! Only one of the following variables will be allocated for a given object.

type(real,1) :: Values1
type(real,2) :: Values2
type(real,3) :: Values3
type(real,4) :: Values4
type(real,5) :: Values5
! Needed for Adaptive Angular Refinement.
! type(Ragged_Right_Real_type) :: ValuesRR
```

```
  end type Collected_Array_type

  ! Global class variables.

  type(integer), parameter :: Version_Increment = 64

contains
```

The Collected_Array Class contains the following routines which are listed in separate sections:

Initialize_Collected_Array (§ D.8.1, page 440)

Finalize_Collected_Array (§ D.8.2, page 445)

Valid_State_Collected_Array (§ D.8.3, page 447)

Initialized_Collected_Array (§ D.8.4, page 449)

Collect_CA_from_OV (§ D.8.5, page 449)

Combine_DV_from_CA (§ D.8.6, page 450)

Gather_and_Collect_CA_from_DV (§ D.8.7, page 453)

Get_Locus_Collected_Array (§ D.8.8, page 455)

Get_Name_Collected_Array (§ D.8.9, page 456)

Get_Values_Collected_Array (§ D.8.10, page 456)

Get_Version_Collected_Array (§ D.8.11, page 457)

Output_Collected_Array (§ D.8.12, page 458)

Set_Values_Collected_Array (§ D.8.13, page 463)

Set_Version_Collected_Array (§ D.8.14, page 464)

end module Caesar_Collected_Array_Class

D.8.1 Initialize_Collected_Array Procedure

The main documentation of the Initialize_Collected_Array Procedure in § 9.8.1 on page 108 contains additional explanation of this code listing.

```
  subroutine Initialize_Collected_Array_1 (CA, OV, Name, status)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variables.

    type(character,*), intent(in), optional :: Name     ! Variable name.
    type(Overlapped_Vector_type), intent(in) :: OV      ! OV for this CA.
```

```
! Output variables.

! Collected_Array to be initialized.
type(Collected_Array_type), intent(inout) :: CA
type(Status_type), intent(out), optional :: status  ! Exit status.

! Internal variables.

type(Status_type), dimension(2) :: allocate_status  ! Allocation Status.
type(Status_type) :: consolidated_status            ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(OV),5)                    ! Overlapped Vector is valid.

! Set up pointers.

CA%One_Structure => OV%One_Structure
CA%Many_Structure => OV%Many_Structure
CA%Many_of_One_Index => OV%Many_of_One_Index

! Set up internals.

if (PRESENT(Name)) CA%Name = Name
CA%Dimensionality = OV%Dimensionality
CA%Version = OV%Version

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)

! Set to maximum dimensionality of 5 to enable VERIFYs without if-checks.
! Also, this gets around problems with using the dimensionality when it
! is set to a negative number for a ragged right array.

call Initialize (CA%Dimensions, 5, allocate_status(1))

! Set initial dimensions.

CA%Dimensions(1:CA%Dimensionality-1) = OV%Dimensions(1:OV%Dimensionality-1)

! Set One Axis size.

CA%Dimensions(CA%Dimensionality) = Length_PE(CA%One_Structure)

! Set Many Axis size.

if (CA%Many_of_One_Index%Dimensionality == 2) then
  CA%A_Dimensionality = CA%Dimensionality + 1
  CA%Dimensions(CA%A_Dimensionality) = &
```

```
    SIZE(CA%Many_of_One_Index%Index2, 2)
else if (CA%Many_of_One_Index%Dimensionality == 1) then
  CA%A_Dimensionality = CA%Dimensionality
  CA%Dimensions(CA%Dimensionality + 1) = 0
else
  ! Shouldn't be triggered. Will add something for RR here later.
  VERIFY(.false.,0)
end if

! Set up the Collected Values.

select case (CA%A_Dimensionality)
case (1)
  call Initialize (CA%Values1, CA%Dimensions(1), allocate_status(2))
  CA%Values1 = OV
case (2)
  call Initialize (CA%Values2, CA%Dimensions(1), CA%Dimensions(2), &
                   allocate_status(2))
  CA%Values2 = OV
case (3)
  call Initialize (CA%Values3, CA%Dimensions(1), CA%Dimensions(2), &
                   CA%Dimensions(3), allocate_status(2))
  CA%Values3 = OV
case (4)
  call Initialize (CA%Values4, CA%Dimensions(1), CA%Dimensions(2), &
                   CA%Dimensions(3), CA%Dimensions(4), allocate_status(2))
  CA%Values4 = OV
case (5)
  call Initialize (CA%Values5, CA%Dimensions(1), CA%Dimensions(2), &
                   CA%Dimensions(3), CA%Dimensions(4), CA%Dimensions(5), &
                   allocate_status(2))
  CA%Values5 = OV
!case (-1)
!  call Initialize (CA%ValuesRR, CA%Dimensions(1), CA%Dimensions(2), &
!                   allocate_status(2))
!  CA%ValuesRR = OV
end select

! Consolidate and handle status.

consolidated_status = allocate_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
  VERIFY(Normal(consolidated_status), 5)
end if
call Finalize (consolidated_status)
call Finalize (allocate_status)

! Set initialization flag.

CA%Initialized = initialized_flag
```

```
    ! Verify guarantees.

    VERIFY(Valid_State(CA),5)  ! CA is now valid.

    return
  end subroutine Initialize_Collected_Array_1

  subroutine Initialize_Collected_Array_2 (CA, Many_of_One_Index, &
                                          Dimensionality, Name, status, &
                                          dim1, dim2, dim3)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variables.

    type(character,*), intent(in), optional :: Name           ! Variable name.
    ! Index for Many-1.
    type(Data_Index_type), intent(in), target :: Many_of_One_Index
    type(integer), intent(in) :: Dimensionality   ! Dimensionality for this CA.
    type(integer), intent(in), optional :: dim1, dim2, dim3   ! Dimensions.

    ! Output variables.

    ! Collected_Array to be initialized.
    type(Collected_Array_type), intent(out) :: CA
    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(2) :: allocate_status ! Allocation Status.
    type(Status_type) :: consolidated_status           ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Many_of_One_Index),5)      ! Index is valid.
    VERIFY(Dimensionality .InInterval. (/1,4/),5) ! Dimensionality is in range.
    VERIFY(PRESENT(dim1) .or. Dimensionality == 1,5) ! Proper dimensions exist.
    VERIFY(PRESENT(dim2) .or. Dimensionality <= 2,5) ! Proper dimensions exist.
    VERIFY(PRESENT(dim3) .or. Dimensionality <= 3,5) ! Proper dimensions exist.

    ! Allocations and initializations.

    call Initialize (allocate_status)
    call Initialize (consolidated_status)

    ! Set up pointers.

    CA%One_Structure => Many_of_One_Index%One_Structure
    CA%Many_Structure => Many_of_One_Index%Many_Structure
    CA%Many_of_One_Index => Many_of_One_Index
```

```
! Set up internals.

if (PRESENT(Name)) CA%Name = Name
CA%Dimensionality = Dimensionality
CA%Version = 0

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)

! Set to maximum dimensionality of 5 to enable VERIFYs without if-checks.
! Also, this gets around problems with using the dimensionality when it
! is set to a negative number for a ragged right array.

call Initialize (CA%Dimensions, 5, allocate_status(1))

! Set initial dimensions.

select case (CA%Dimensionality)
case (1)
  ! No initial dimensions.
case (2)
  CA%Dimensions(1) = dim1
case (3)
  CA%Dimensions(1) = dim1
  CA%Dimensions(2) = dim2
case (4)
  CA%Dimensions(1) = dim1
  CA%Dimensions(2) = dim2
  CA%Dimensions(3) = dim3
end select

! Set One Axis size.

CA%Dimensions(CA%Dimensionality) = Length_PE(CA%One_Structure)

! Set Many Axis size.

if (CA%Many_of_One_Index%Dimensionality == 2) then
  CA%A_Dimensionality = CA%Dimensionality + 1
  CA%Dimensions(CA%A_Dimensionality) = &
    SIZE(CA%Many_of_One_Index%Index2, 2)
else if (CA%Many_of_One_Index%Dimensionality == 1) then
  CA%A_Dimensionality = CA%Dimensionality
  CA%Dimensions(CA%Dimensionality + 1) = 0
else
  ! Shouldn't be triggered. Will add something for RR here later.
  VERIFY(.false.,0)
end if

! Set up the Collected Values.
```

```
    select case (CA%A_Dimensionality)
    case (1)
      call Initialize (CA%Values1, CA%Dimensions(1), allocate_status(2))
    case (2)
      call Initialize (CA%Values2, CA%Dimensions(1), CA%Dimensions(2), &
                       allocate_status(2))
    case (3)
      call Initialize (CA%Values3, CA%Dimensions(1), CA%Dimensions(2), &
                       CA%Dimensions(3), allocate_status(2))
    case (4)
      call Initialize (CA%Values4, CA%Dimensions(1), CA%Dimensions(2), &
                       CA%Dimensions(3), CA%Dimensions(4), allocate_status(2))
    case (5)
      call Initialize (CA%Values5, CA%Dimensions(1), CA%Dimensions(2), &
                       CA%Dimensions(3), CA%Dimensions(4), CA%Dimensions(5), &
                       allocate_status(2))
    !case (-1)
    !  call Initialize (CA%ValuesRR, CA%Dimensions(1), CA%Dimensions(2), &
    !                   allocate_status(2))
    end select

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set initialization flag.

    CA%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(CA),5)  ! CA is now valid.

    return
  end subroutine Initialize_Collected_Array_2
```

D.8.2 Finalize_Collected_Array Procedure

The main documentation of the Finalize_Collected_Array Procedure in § 9.8.2 on page 109 contains additional explanation of this code listing.

```
  subroutine Finalize_Collected_Array (CA, status)

    ! Use associations.
```

```
use Caesar_Flags_Module, only: uninitialized_flag

! Input/Output variable.

! Collected_Array to be finalized.
type(Collected_Array_type), intent(inout) :: CA

! Output variable.

type(Status_type), intent(out), optional :: status   ! Exit status.

! Internal variables.

type(Status_type), dimension(6) :: deallocate_status ! Deallocation Status.
type(Status_type) :: consolidated_status             ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(CA),7) ! CA is valid.

! Unset initialization flag.

CA%Initialized = uninitialized_flag

! Set deallocation status.

call Initialize (deallocate_status)
call Initialize (consolidated_status)

! Finalize internals.

NULLIFY(CA%One_Structure)
NULLIFY(CA%Many_Structure)
NULLIFY(CA%Many_of_One_Index)

select case (CA%A_Dimensionality)
case (1)
  call Finalize (CA%Values1, deallocate_status(1))
case (2)
  call Finalize (CA%Values2, deallocate_status(1))
case (3)
  call Finalize (CA%Values3, deallocate_status(1))
case (4)
  call Finalize (CA%Values4, deallocate_status(1))
case (5)
  call Finalize (CA%Values5, deallocate_status(1))
!case (-1)
!  call Finalize (CA%ValuesRR, deallocate_status(1))
end select
call Finalize (CA%A_Dimensionality, deallocate_status(2))
call Finalize (CA%Dimensionality,   deallocate_status(3))
```

```
    call Finalize (CA%Dimensions,       deallocate_status(4))
    call Finalize (CA%Name,             deallocate_status(5))
    call Finalize (CA%Version,          deallocate_status(6))

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(CA),7) ! CA is not valid.

    return
  end subroutine Finalize_Collected_Array
```

D.8.3 Valid_State_Collected_Array Procedure

The main documentation of the Valid_State_Collected_Array Procedure in § 9.8.3 on page 110 contains additional explanation of this code listing.

```
  function Valid_State_Collected_Array (CA) result(Valid)

    ! Input variables.

    ! Variable to be checked.
    type(Collected_Array_type), intent(in) :: CA

    ! Output variables.

    type(logical) :: Valid       ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for association of pointered internals.

    Valid = Valid .and. ASSOCIATED(CA%One_Structure)
    Valid = Valid .and. ASSOCIATED(CA%Many_Structure)
    Valid = Valid .and. ASSOCIATED(CA%Dimensions)
    Valid = Valid .and. ASSOCIATED(CA%Many_of_One_Index)
    if (.not.Valid) return
```

```
  ! Check for validity of internals.

  Valid = Valid .and. Initialized(CA)
  Valid = Valid .and. Valid_State(CA%A_Dimensionality)
  Valid = Valid .and. Valid_State(CA%Dimensionality)
  Valid = Valid .and. Valid_State(CA%Dimensions)
  Valid = Valid .and. Valid_State(CA%Many_Structure)
  Valid = Valid .and. Valid_State(CA%Many_of_One_Index)
  Valid = Valid .and. Valid_State(CA%Name)
  Valid = Valid .and. Valid_State(CA%One_Structure)
  select case (CA%A_Dimensionality)
  case (1)
    Valid = Valid .and. Valid_State(CA%Values1)
  case (2)
    Valid = Valid .and. Valid_State(CA%Values2)
  case (3)
    Valid = Valid .and. Valid_State(CA%Values3)
  case (4)
    Valid = Valid .and. Valid_State(CA%Values4)
  case (5)
    Valid = Valid .and. Valid_State(CA%Values5)
  !case (-1)
  !  Valid = Valid .and. Valid_State(CA%ValuesRR)
  end select
  Valid = Valid .and. Valid_State(CA%Version)
  if (.not.Valid) return

  ! Checks on the validity of CA.

  Valid = Valid .and. CA%A_Dimensionality == &
          CA%Dimensionality + CA%Many_of_One_Index%Dimensionality - 1
  select case (CA%A_Dimensionality)
  case (1)
    Valid = Valid .and. ALL(CA%Dimensions(1:1) == SHAPE(CA%Values1))
  case (2)
    Valid = Valid .and. ALL(CA%Dimensions(1:2) == SHAPE(CA%Values2))
  case (3)
    Valid = Valid .and. ALL(CA%Dimensions(1:3) == SHAPE(CA%Values3))
  case (4)
    Valid = Valid .and. ALL(CA%Dimensions(1:4) == SHAPE(CA%Values4))
  case (5)
    Valid = Valid .and. ALL(CA%Dimensions(1:5) == SHAPE(CA%Values5))
  !case (-1)
  !  Valid = Valid .and. ALL(CA%Dimensions(1:2) == SHAPE(CA%ValuesRR))
  end select

  return
end function Valid_State_Collected_Array
```

D.8.4 Initialized_Collected_Array Procedure

The main documentation of the Initialized_Collected_Array Procedure in § 9.8.4 on page 110 contains additional explanation of this code listing.

```
function Initialized_Collected_Array (CA) result(Initialized)

  ! Use associations.

  use Caesar_Flags_Module, only: initialized_flag

  ! Input variable.

  ! Collected_Array to be checked.
  type(Collected_Array_type), intent(in) :: CA

  ! Output variable.

  type(logical) :: Initialized          ! Initialized condition boolean.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements - none.

  ! Set initialized boolean.

  Initialized = CA%Initialized == initialized_flag

  ! Verify guarantees - none.

  return
end function Initialized_Collected_Array
```

D.8.5 Collect_CA_from_OV Procedure

The main documentation of the Collect_CA_from_OV Procedure in § 9.8.5 on page 110 contains additional explanation of this code listing.

```
subroutine Collect_CA_from_OV (CA, OV)

  ! Input variable.

  type(Overlapped_Vector_type), intent(in) :: OV  ! OV to be collected.

  ! Input/Output variable.

  type(Collected_Array_type), intent(inout) :: CA ! Resultant CA.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.
```

```
  VERIFY(Valid_State(CA),5)                          ! CA is valid.
  VERIFY(Valid_State(OV),5)                          ! OV is valid.
  VERIFY(CA%Dimensionality == OV%Dimensionality,5)   ! Same dimensionality.
  VERIFY(CA%Dimensions(1:CA%Dimensionality-1) == dnl
         OV%Dimensions(1:OV%Dimensionality-1),5)     ! Same dimensions.
  ! Same structures and indices.
  VERIFY(ASSOCIATED(CA%One_Structure,OV%One_Structure),5)
  VERIFY(ASSOCIATED(CA%Many_Structure,OV%Many_Structure),5)
  VERIFY(ASSOCIATED(CA%Many_of_One_Index,OV%Many_of_One_Index),5)

  ! Collect the values, using the Collect_and_Access OV routine.

  select case (CA%A_Dimensionality)
  case (1)
    CA%Values1 = OV
  case (2)
    CA%Values2 = OV
  case (3)
    CA%Values3 = OV
  case (4)
    CA%Values4 = OV
  case (5)
    CA%Values5 = OV
  end select

  ! Set version number.

  CA = Version(OV)

  ! Verify guarantees.

  VERIFY(Valid_State(CA),5)    ! CA is valid.
  VERIFY(Valid_State(OV),5)    ! OV is still valid.

  return
end subroutine Collect_CA_from_OV
```

D.8.6 Combine_DV_from_CA Procedure

The main documentation of the Combine_DV_from_CA Procedure in § 9.8.6 on page 111 contains additional explanation of this code listing.

```
define([COMBINE_ROUTINE],[
  pushdef([OP], [$1])
  pushdef([Combine_with_OP_DV_from_CA], expand(Combine_with_$1_DV_from_CA))

  subroutine Combine_with_OP_DV_from_CA (DV, CA)

    ! Input variable.

    type(Collected_Array_type), intent(in) :: CA ! Variable to be combined.
```

```
! Input/Output variable.

type(Distributed_Vector_type), intent(inout) :: DV ! Resultant DV.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(CA),5)                            ! CA is valid.
VERIFY(Valid_State(DV),5)                            ! DV is valid.
VERIFY(CA%Dimensionality == DV%Dimensionality,5)    ! Same dimensionality.
VERIFY(CA%Dimensions(1:CA%Dimensionality) == dnl
       DV%Dimensions(1:DV%Dimensionality),5)        ! Same dimensions.
VERIFY(ASSOCIATED(CA%One_Structure,DV%Structure),5) ! Same one-structure.

! Combine the values. There are different versions based on the
! dimensionality of the Index and on the dimensionality of the "Array"
! itself.

select case (CA%Many_of_One_Index%Dimensionality)

! Vector Index - no combination necessary.

case (1)

  ! Switch on the dimensionality of the data itself.

  select case (CA%Dimensionality)
  case (1)
    DV%Values1 = CA%Values1
  case (2)
    DV%Values2 = CA%Values2
  case (3)
    DV%Values3 = CA%Values3
  case (4)
    DV%Values4 = CA%Values4
  end select

! Array Index. Shape of CA%Values must be:
!
!     CA%Values ( [dim1, [dim2, [dim3, ]]] One_Axis, Many_Axis )

case (2)

  ! Switch on the dimensionality of the data itself.

  select case (CA%Dimensionality)

  case (1)

    DV%Values1(:) = OPERATION(CA%Values2, 2)
    ifelse(OP, [Average], [
      DV%Values1 = DV%Values1 / changetype(real,SIZE(CA%Values2, 2))
```

```
        ])

      case (2)

        DV%Values2(:,:) = OPERATION(CA%Values3, 3)
        ifelse(OP, [Average], [
          DV%Values2 = DV%Values2 / changetype(real,SIZE(CA%Values3, 3))
        ])

      case (3)

        DV%Values3(:,:,:) = OPERATION(CA%Values4, 4)
        ifelse(OP, [Average], [
          DV%Values3 = DV%Values3 / changetype(real,SIZE(CA%Values4, 4))
        ])

      case (4)

        DV%Values4(:,:,:,:) = OPERATION(CA%Values5, 5)
        ifelse(OP, [Average], [
          DV%Values4 = DV%Values4 / changetype(real,SIZE(CA%Values5, 5))
        ])

      end select

    end select

    ! Set version number.

    DV = Version(CA)

    ! Verify guarantees.

    VERIFY(Valid_State(CA),5)    ! CA is still valid.
    VERIFY(Valid_State(DV),5)    ! DV is valid.

    return
  end subroutine Combine_with_OP_DV_from_CA

  popdef([OP])
  popdef([OPERATION])
  popdef([Combine_with_OP_DV_from_CA])
])

! Add "Conserve" later if needed.

fortext([Op],[Average SUM MAX MIN],[
  ifelse(
    Op, [MAX], [
      pushdef([OPERATION], [MAXVAL[]($1, $2)])
    ], Op, [MIN], [
      pushdef([OPERATION], [MINVAL[]($1, $2)])
    ], [
      pushdef([OPERATION], [SUM[]($1, $2)])
```

```
    ]
  )
  COMBINE_ROUTINE(Op)
])
```

D.8.7 Gather_and_Collect_CA_from_DV Procedure

The main documentation of the Gather_and_Collect_CA_from_DV Procedure in § 9.8.7 on page 111 contains additional explanation of this code listing.

```
subroutine Gather_and_Collect_CA_from_DV (CA, DV)

  ! Input variable.

  type(Distributed_Vector_type), intent(in) :: DV     ! DV to be gathered.

  ! Input/Output variable.

  type(Collected_Array_type), intent(inout) :: CA     ! Gathered vector.

  ! Internal variables.

  type(integer) :: i, j, k                            ! Loop counters.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(CA),5)                        ! CA is valid.
  VERIFY(Valid_State(DV),5)                        ! DV is valid.
  ! DV, CA -> same Many Structure.
  VERIFY(ASSOCIATED(DV%Structure,CA%Many_Structure),5)
  ! DV, CA -> same Dimensionality and Dimensions.
  VERIFY(DV%Dimensionality == CA%Dimensionality,5)
  VERIFY(CA%Dimensions(1:CA%Dimensionality-2) == dnl
         DV%Dimensions(1:DV%Dimensionality-2),5)    ! Same dimensions.

  ! Gather the off PE values if they are out-of-date. This could be done
  ! faster for multiple dimensions by packing everything into a single
  ! vector, gathering, and then unpacking. Maybe in a later version.

  if (Version(CA) /= Version(DV)) then

    select case (CA%Many_of_One_Index%Dimensionality)

    ! Vector Index.

    case (1)

      select case (CA%Dimensionality)
      case (1)
        call Gather (CA%Values1, DV%Values1, Trace=CA%Many_of_One_Index%Trace)
```

```
    case (2)
      do i = 1, CA%Dimensions(1)
        call Gather (CA%Values2(i,:), DV%Values2(i,:), &
                     Trace=CA%Many_of_One_Index%Trace)
      end do
    case (3)
      do i = 1, CA%Dimensions(1)
        do j = 1, CA%Dimensions(2)
          call Gather (CA%Values3(i,j,:), DV%Values3(i,j,:), &
                       Trace=CA%Many_of_One_Index%Trace)
        end do
      end do
    case (4)
      do i = 1, CA%Dimensions(1)
        do j = 1, CA%Dimensions(2)
          do k = 1, CA%Dimensions(3)
            call Gather (CA%Values4(i,j,k,:), DV%Values4(i,j,k,:), &
                         Trace=CA%Many_of_One_Index%Trace)
          end do
        end do
      end do
    !case (-1)
    !  call Gather (CA%ValuesRR, DV%ValuesRR, &
    !               Trace=CA%Many_of_One_Index%Trace)
    end select

  ! Array Index.

  case (2)

    select case (CA%Dimensionality)
    case (1)
      call Gather (CA%Values2, DV%Values1, Trace=CA%Many_of_One_Index%Trace)
    case (2)
      do i = 1, CA%Dimensions(1)
        call Gather (CA%Values3(i,:,:), DV%Values2(i,:), &
                     Trace=CA%Many_of_One_Index%Trace)
      end do
    case (3)
      do i = 1, CA%Dimensions(1)
        do j = 1, CA%Dimensions(2)
          call Gather (CA%Values4(i,j,:,:), DV%Values3(i,j,:), &
                       Trace=CA%Many_of_One_Index%Trace)
        end do
      end do
    case (4)
      do i = 1, CA%Dimensions(1)
        do j = 1, CA%Dimensions(2)
          do k = 1, CA%Dimensions(3)
            call Gather (CA%Values5(i,j,k,:,:), DV%Values4(i,j,k,:), &
                         Trace=CA%Many_of_One_Index%Trace)
          end do
        end do
      end do
```

```
        !case (-1)
        !  call Gather (CA%ValuesRR, DV%ValuesRR, &
        !              Trace=CA%Many_of_One_Index%Trace)
        end select

      end select

    end if

    ! Set version.

    CA = Version(DV)

    ! Verify guarantees.

    VERIFY(Valid_State(CA),5)  ! CA is valid.

    return
  end subroutine Gather_and_Collect_CA_from_DV
```

D.8.8 Get_Locus_Collected_Array Procedure

The main documentation of the Get_Locus_Collected_Array Procedure in § 9.8.8 on page 112 contains additional explanation of this code listing.

```
  define([LOCUS_ROUTINE],[
    pushdef([Get_MANYORONE_Locus_CA], expand(Get_$1_Locus_CA))

    function Get_MANYORONE_Locus_CA (CA) result(Locus_CA)

      ! Input variable.

      type(Collected_Array_type), intent(in) :: CA   ! Variable to be queried.

      ! Output variable.

      type(character,name_length) :: Locus_CA       ! Locus of CA.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(CA),5)      ! CA is valid.

      ! Set the value.

      Locus_CA = Locus(CA%$1_Structure)

      ! Verify guarantees - none.

      return
    end function Get_MANYORONE_Locus_CA
```

```
  popdef([Get_MANYORONE_Locus_CA])
])

LOCUS_ROUTINE(Many)
LOCUS_ROUTINE(One)
```

D.8.9 Get_Name_Collected_Array Procedure

The main documentation of the Get_Name_Collected_Array Procedure in § 9.8.9 on page 112 contains additional explanation of this code listing.

```
function Get_Name_Collected_Array (CA) result(Name)

  ! Input variable.

  type(Collected_Array_type), intent(in) :: CA  ! Variable to be queried.

  ! Output variable.

  type(character,name_length) :: Name             ! Name of CA.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(CA),5)      ! CA is valid.

  ! Set the value.

  Name = CA%Name

  ! Verify guarantees - none.

  return
end function Get_Name_Collected_Array
```

D.8.10 Get_Values_Collected_Array Procedure

The main documentation of the Get_Values_Collected_Array Procedure in § 9.8.10 on page 112 contains additional explanation of this code listing.

```
define([GET_VALUES_ROUTINE],[
  pushdef([DIM], [$1])
  pushdef([ValuesDIM], expand(Values[]DIM))
  pushdef([Get_Values_Collected_Array_DIM],
    expand(Get_Values_Collected_Array_DIM))

  subroutine Get_Values_Collected_Array_DIM (Values, CA)
```

```
      ! Input variable.

      type(Collected_Array_type), intent(in) :: CA ! Variable to be queried.

      ! Input/Output variable.

      type(real,DIM,np), intent(inout) :: Values   ! Values bare naked array.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(CA),5)                      ! CA is valid.
      VERIFY(Valid_State_NP(Values),5)               ! Values is valid.
      VERIFY(DIM == CA%A_Dimensionality,5)   ! CA has been set up for this call.
      VERIFY(SHAPE(Values) == SHAPE(CA%ValuesDIM),5) ! Values shape check.

      ! Set the values.

      Values = CA%ValuesDIM

      ! Verify guarantees.

      VERIFY(Valid_State(CA),5)          ! CA is still valid.
      VERIFY(Valid_State_NP(Values),5)   ! Values is valid.

      return
    end subroutine Get_Values_Collected_Array_DIM

    popdef([DIM])
    popdef([ValuesDIM])
    popdef([Get_Values_Collected_Array_DIM])
  ])

  forloop([Dim],[1],[5],[
    GET_VALUES_ROUTINE(Dim)
  ])
```

D.8.11 Get_Version_Collected_Array Procedure

The main documentation of the Get_Version_Collected_Array Procedure in § 9.8.11 on page 113 contains additional explanation of this code listing.

```
  function Get_Version_Collected_Array (CA) result(Version)

    ! Input variable.

    type(Collected_Array_type), intent(in) :: CA   ! Variable to be queried.

    ! Output variable.
```

```
    type(integer) :: Version                       ! Version number.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(CA),5)      ! CA is valid.

    ! Get the value.

    Version = CA%Version

    ! Verify guarantees - none.

    return
  end function Get_Version_Collected_Array
```

D.8.12 Output_Collected_Array Procedure

The main documentation of the Output_Collected_Array Procedure in § 9.8.12 on page 113 contains additional explanation of this code listing.

```
  subroutine Output_Collected_Array (CA, One_First, One_Last, Unit)

    ! Input variables.

    type(Collected_Array_type), intent(in) :: CA      ! Variable to be output.
    type(integer), intent(in), optional :: One_First  ! Extents of one value
    type(integer), intent(in), optional :: One_Last   !   data to be output.
    type(integer), intent(in), optional :: Unit       ! Output unit.

    ! Internal variables.

    type(integer) :: Buffer_Loc                     ! Buffer location.
    type(integer) :: Buffer_Size                    ! Output buffer size.
    type(integer) :: Buffer_Skip                    ! Buffer increment.
    type(integer) :: i_local, i_global              ! Loop counter.
    type(integer) :: A_One_First                    ! Actual one first value.
    type(integer) :: A_One_Last                     ! Actual one last value.
    type(integer) :: A_Unit                         ! Actual output unit.
    type(character,80) :: Name_Name                 ! Name of the CA.
    type(character,80) :: Output_1                  ! Output buffer.
    type(character,80,1) :: Output_Buffer           ! Output buffer vector.
    type(integer) :: Version_Number                 ! Version of the CA.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(CA),5)      ! CA is valid.

    ! Set unit number.
```

```
if (PRESENT(Unit)) then
  A_Unit = Unit
else
  A_Unit = 6
end if

! These are evaluated on all PEs -- NOT inside an IO PE block -- because
! they contain validity checks on CA and thus require global communication.

Version_Number = Version(CA)
Name_Name = Name(CA)

! Output Identification Info.

if (this_is_IO_PE) then
  write (A_Unit,100) 'Collected Array Information:'
  write (A_Unit,*) ' Name                    = ', TRIM(Name_Name)
  write (A_Unit,*) ' Locus                   = ', &
                   TRIM(Locus(CA%Many_Structure)), ' of ', &
                   TRIM(Locus(CA%One_Structure))
  write (A_Unit,*) ' Initialized             = ', Initialized(CA)
  write (A_Unit,*) ' Version                 = ', Version_Number
  write (A_Unit,*) ' Dimensionality          = ', CA%Dimensionality
  write (A_Unit,*) ' A_Dimensionality        = ', CA%A_Dimensionality
end if

! PE-dependent info.

write (Output_1,101) 'PE:', this_PE, &
                     ', Dimensions    =', CA%Dimensions
call Parallel_Write (Output_1, A_Unit)

! Set up actual limit values for pass-through.

if (PRESENT(One_First)) then
  A_One_First = One_First
else
  A_One_First = 1
end if
if (PRESENT(One_Last)) then
  A_One_Last = One_Last
else
  A_One_Last = Length_Total(CA%One_Structure)
end if
A_One_First = MAX(A_One_First, First_PE(CA%One_Structure))
A_One_Last = MIN(A_One_Last, Last_PE(CA%One_Structure))

! Output internal structure info.

call Output (CA%Many_of_One_Index, A_One_First, A_One_Last, A_Unit, &
             Indent=2)

! Output internal values.
```

```
if (this_is_IO_PE) then
  write (A_Unit,100) '  Internal Values:'
end if

! Output the values based on the dimensionality.

select case (CA%Many_of_One_Index%Dimensionality)

! Vector index.

case (1)

  select case (CA%Dimensionality)
  case (1)
    Buffer_Size = MAX(0, (A_One_Last - A_One_First + 1))
    call Initialize (Output_Buffer, Buffer_Size)
    if (Buffer_Size /= 0) then
      Buffer_Skip = 1
      Buffer_Loc = 1
      do i_global = A_One_First, A_One_Last
        i_local = i_global - First_PE(CA%One_Structure) + 1
        write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
          'PE:', this_PE, ', Values1(', i_global, ') =', &
          CA%Values1(i_local)
        Buffer_Loc = Buffer_Loc + Buffer_Skip
      end do
    end if
  case (2)
    Buffer_Size = MAX(0, ((SIZE(CA%Values2(:,1)) + 2) / 3) &
                  * (A_One_Last - A_One_First + 1))
    call Initialize (Output_Buffer, Buffer_Size)
    if (Buffer_Size /= 0) then
      Buffer_Skip = (SIZE(CA%Values2(:,1)) + 2) / 3
      Buffer_Loc = 1
      do i_global = A_One_First, A_One_Last
        i_local = i_global - First_PE(CA%One_Structure) + 1
        write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
          'PE:', this_PE, ', Values2(:,', i_global, ') =', &
          CA%Values2(:,i_local)
        Buffer_Loc = Buffer_Loc + Buffer_Skip
      end do
    end if
  case (3)
    Buffer_Size = MAX(0, ((SIZE(CA%Values3(:,:,1)) + 2) / 3) &
                  * (A_One_Last - A_One_First + 1))
    call Initialize (Output_Buffer, Buffer_Size)
    if (Buffer_Size /= 0) then
      Buffer_Skip = (SIZE(CA%Values3(:,:,1)) + 2) / 3
      Buffer_Loc = 1
      do i_global = A_One_First, A_One_Last
        i_local = i_global - First_PE(CA%One_Structure) + 1
        write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
          'PE:', this_PE, ', Values3(:,:,', i_global, ') =', &
```

```
              CA%Values3(:,:,i_local)
            Buffer_Loc = Buffer_Loc + Buffer_Skip
          end do
        end if
      case (4)
        Buffer_Size = MAX(0, ((SIZE(CA%Values4(:,:,:,1)) + 2) / 3) &
                      * (A_One_Last - A_One_First + 1))
        call Initialize (Output_Buffer, Buffer_Size)
        if (Buffer_Size /= 0) then
          Buffer_Skip = (SIZE(CA%Values4(:,:,:,1)) + 2) / 3
          Buffer_Loc = 1
          do i_global = A_One_First, A_One_Last
            i_local = i_global - First_PE(CA%One_Structure) + 1
            write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
              'PE:', this_PE, ', Values4(:,:,:,', i_global, ') =', &
              CA%Values4(:,:,:,i_local)
            Buffer_Loc = Buffer_Loc + Buffer_Skip
          end do
        end if
!      case (-1)
!        Buffer_Size = MAX(0, ((SIZE(CA%ValuesRR(:,1)) + 2) / 3) &
!                      * (A_One_Last - A_One_First + 1))
!        call Initialize (Output_Buffer, Buffer_Size)
!        if (Buffer_Size /= 0) then
!          Buffer_Skip = (SIZE(CA%ValuesRR(:,1)) + 2) / 3
!          Buffer_Loc = 1
!          do i_global = A_One_First, A_One_Last
!            i_local = i_global - First_PE(CA%One_Structure) + 1
!            write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
!              'PE:', this_PE, ', ValuesRR(:,', i_global, ') =', &
!              CA%ValuesRR(:,i_local)
!            Buffer_Loc = Buffer_Loc + Buffer_Skip
!          end do
!        end if
      end select

    ! Array index.

    case (2)

      select case (CA%Dimensionality)
      case (1)
        Buffer_Size = MAX(0, (A_One_Last - A_One_First + 1))
        call Initialize (Output_Buffer, Buffer_Size)
        if (Buffer_Size /= 0) then
          Buffer_Skip = 1
          Buffer_Loc = 1
          do i_global = A_One_First, A_One_Last
            i_local = i_global - First_PE(CA%One_Structure) + 1
            write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
              'PE:', this_PE, ', Values2(', i_global, ',:) =', &
              CA%Values2(i_local,:)
            Buffer_Loc = Buffer_Loc + Buffer_Skip
          end do
```

```
        end if
      case (2)
        Buffer_Size = MAX(0, ((SIZE(CA%Values3(:,1,:)) + 2) / 3) &
                      * (A_One_Last - A_One_First + 1))
        call Initialize (Output_Buffer, Buffer_Size)
        if (Buffer_Size /= 0) then
          Buffer_Skip = (SIZE(CA%Values3(:,1,:)) + 2) / 3
          Buffer_Loc = 1
          do i_global = A_One_First, A_One_Last
            i_local = i_global - First_PE(CA%One_Structure) + 1
            write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
              'PE:', this_PE, ', Values3(:,', i_global, ',:) =', &
              CA%Values3(:,i_local,:)
            Buffer_Loc = Buffer_Loc + Buffer_Skip
          end do
        end if
      case (3)
        Buffer_Size = MAX(0, ((SIZE(CA%Values4(:,:,1,:)) + 2) / 3) &
                      * (A_One_Last - A_One_First + 1))
        call Initialize (Output_Buffer, Buffer_Size)
        if (Buffer_Size /= 0) then
          Buffer_Skip = (SIZE(CA%Values4(:,:,1,:)) + 2) / 3
          Buffer_Loc = 1
          do i_global = A_One_First, A_One_Last
            i_local = i_global - First_PE(CA%One_Structure) + 1
            write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
              'PE:', this_PE, ', Values4(:,:,', i_global, ',:) =', &
              CA%Values4(:,:,i_local,:)
            Buffer_Loc = Buffer_Loc + Buffer_Skip
          end do
        end if
      case (4)
        Buffer_Size = MAX(0, ((SIZE(CA%Values5(:,:,:,1,:)) + 2) / 3) &
                      * (A_One_Last - A_One_First + 1))
        call Initialize (Output_Buffer, Buffer_Size)
        if (Buffer_Size /= 0) then
          Buffer_Skip = (SIZE(CA%Values5(:,:,:,1,:)) + 2) / 3
          Buffer_Loc = 1
          do i_global = A_One_First, A_One_Last
            i_local = i_global - First_PE(CA%One_Structure) + 1
            write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
              'PE:', this_PE, ', Values5(:,:,:,', i_global, ',:) =', &
              CA%Values5(:,:,:,i_local,:)
            Buffer_Loc = Buffer_Loc + Buffer_Skip
          end do
        end if
!      case (-1)
!        Buffer_Size = MAX(0, ((SIZE(CA%ValuesRR(:,1)) + 2) / 3) &
!                      * (A_One_Last - A_One_First + 1))
!        call Initialize (Output_Buffer, Buffer_Size)
!        if (Buffer_Size /= 0) then
!          Buffer_Skip = (SIZE(CA%ValuesRR(:,1)) + 2) / 3
!          Buffer_Loc = 1
!          do i_global = A_One_First, A_One_Last
```

```
!           i_local = i_global - First_PE(CA%One_Structure) + 1
!           write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),102) &
!             'PE:', this_PE, ', ValuesRR(:,', i_global, ',:) =', &
!             CA%ValuesRR(:,i_local)
!           Buffer_Loc = Buffer_Loc + Buffer_Skip
!         end do
!       end if
      end select

    end select

    call Parallel_Write (Output_Buffer, A_Unit)
    call Finalize (Output_Buffer)

    ! Format statements. With these formats, this should work up to
    ! (10^6 - 1) PEs.

100 format (/, a, /)
101 format (2x, a, i5, a, i9, :, 4(',', i9, :))
102 format (2x, a, i5, a, i11, a, 1p, e13.5e3, :, &
            2(',', e13.5e3, :), ',', /, &
            (38x, e13.5e3, :, 2(',', e13.5e3, :), ','))

    ! Verify guarantees - none.

    return
  end subroutine Output_Collected_Array
```

D.8.13 Set_Values_Collected_Array Procedure

The main documentation of the Set_Values_Collected_Array Procedure in § 9.8.13 on page 113 contains additional explanation of this code listing.

```
  define([SET_VALUES_ROUTINE],[
    pushdef([DIM], [$1])
    pushdef([Set_Values_Collected_Array_DIM],
      expand(Set_Values_Collected_Array_DIM))

    subroutine Set_Values_Collected_Array_DIM (CA, Values)

      ! Input variable.

      type(real,DIM,np), intent(in) :: Values       ! Values bare naked array.

      ! Input/Output variable.

      type(Collected_Array_type), intent(inout) :: CA ! Variable to be set.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.
```

```
    VERIFY(Valid_State(CA),5)                      ! CA is valid.
    VERIFY(Valid_State_NP(Values),5)               ! Values is valid.
    VERIFY($1 == CA%A_Dimensionality,5)   ! CA has been set up for this call.
    VERIFY(SHAPE(Values) == SHAPE(CA%Values$1),5)  ! Values shape check.

    ! Set the values.

    CA%Values$1 = Values

    ! Increment the version number.

    CA%Version = CA%Version + Version_Increment

    ! Verify guarantees.

    VERIFY(Valid_State(CA),5)                            ! CA is still valid.

    return
  end subroutine Set_Values_Collected_Array_DIM

  popdef([DIM])
  popdef([Set_Values_Collected_Array_DIM])
])

forloop([Dim],[1],[5],[
  SET_VALUES_ROUTINE(Dim)
])
```

D.8.14 Set_Version_Collected_Array Procedure

The main documentation of the Set_Version_Collected_Array Procedure in § 9.8.14 on page 114 contains additional explanation of this code listing.

```
subroutine Set_Version_Collected_Array (CA, Version)

  ! Input variable.

  type(integer), intent(in) :: Version            ! Version number.

  ! Input/Output variable.

  type(Collected_Array_type), intent(inout) :: CA  ! Variable to be set.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(CA),5)      ! CA is valid.

  ! Set the value.

  CA%Version = Version
```

```
    ! Verify guarantees - none.

    return
  end subroutine Set_Version_Collected_Array
```

D.8.15 Collected_Array Class Unit Test Program

This lightly commented program performs a unit test on the Collected_Array Class, which is described in § 9.8 on page 107.

```
program Unit_Test
  use Caesar_Intrinsics_Module
  use Caesar_Base_Structure_Class
  use Caesar_Data_Index_Class
  use Caesar_Assembled_Vector_Class
  use Caesar_Distributed_Vector_Class
  use Caesar_Overlapped_Vector_Class
  use Caesar_Collected_Array_Class
  use Caesar_Communication_Class
  use Caesar_Numbers_Module, only: one, four, zero
  implicit none

  type(Communication_type) :: Comm
  type(Base_Structure_type) :: Cell_Structure, Node_Structure
  type(integer,2) :: Nodes_of_Cells_Index_Values
  type(Data_Index_type) :: Nodes_of_Cells_Index
  type(Assembled_Vector_type) :: Coordinates_Nodes_AV
  type(Distributed_Vector_type) :: Coordinates_Nodes_DV, Results_Cells_DV
  type(Overlapped_Vector_type) :: Coordinates_Nodes_of_Cells_OV
  type(Collected_Array_type) :: Coordinates_Nodes_of_Cells_CA
  type(Status_type) :: status
  type(character,name_length) :: Name_Name
  type(real,2) :: Coordinates, Results_Cells_BNV
  type(real,3) :: Processed_Coordinates
  type(integer) :: NodeSize, Dimensionality, DimSize, &
                   Many_Axis_Length, n, NDimensions, Nodes_per_Cell, &
                   NNodes
  type(logical) :: detailed_output, Success

  ! Initializations.

  call Initialize (Comm)
  call Output (Comm)
  call Initialize (status)
  call Initialize (Name_Name)
  Dimensionality = 2
  NDimensions = 2
  detailed_output = NPEs <= 8

  ! Set up the Shell Partition Structures.
```

```
call Initialize_Shell_Partition (NDimensions, Cell_Structure, &
                                 Node_Structure, Nodes_of_Cells_Index, &
                                 detailed_output)

! Initialize AV, DV, OV and CA Coordinate vectors.

Name_Name = 'Coordinates of Nodes'
call Initialize (Coordinates_Nodes_AV, Node_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)
call Initialize (Coordinates_Nodes_DV, Node_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)
Name_Name = 'Coordinates of Nodes of Cells'
call Initialize (Coordinates_Nodes_of_Cells_OV, Nodes_of_Cells_Index, &
                 Dimensionality, Name_Name, status, NDimensions)
call Initialize (Coordinates_Nodes_of_Cells_CA, &
                 Coordinates_Nodes_of_Cells_OV, Name_Name, status)
Name_Name = 'Results of Cells'
call Initialize (Results_Cells_DV, Cell_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)

! Set up Coordinates array on IO PE only.

NNodes = Length_Total(Node_Structure)
if (this_is_IO_PE) then
  DimSize = NDimensions
  NodeSize = NNodes
else
  DimSize = 0
  NodeSize = 0
end if
call Initialize (Coordinates, DimSize, NodeSize)
if (this_is_IO_PE) then
  Coordinates(1,:) = (/ ( changetype(real,(n)), n = 1,NNodes ) /)
  Coordinates(2,:) = (/ ( one, n = 1,NNodes ) /)
end if

Name_Name = ''

! Set up Processed Coordinates array on every processor.

Nodes_per_Cell = 2**NDimensions
Many_Axis_Length = Nodes_per_Cell
call Initialize (Processed_Coordinates, NDimensions, &
                 Length_PE(Cell_Structure), Many_Axis_Length)

! Set up Results_Cells_BNV array on every processor.

Nodes_per_Cell = 2**NDimensions
call Initialize (Results_Cells_BNV, NDimensions, &
                 Length_PE(Cell_Structure))

! Version number check.

Coordinates_Nodes_of_Cells_CA = 123
```

```
  Success = Version(Coordinates_Nodes_of_Cells_CA) == 123
  call Output_Test ('Version number', Success)

  ! Send Coordinates into Assembled Vector, then Distributed Vector,
  ! then Collected Array, and access the data.

  Coordinates_Nodes_AV = Coordinates
  Coordinates_Nodes_DV = Coordinates_Nodes_AV
  Coordinates_Nodes_of_Cells_CA = Coordinates_Nodes_DV
  Processed_Coordinates = Coordinates_Nodes_of_Cells_CA

  ! Re-construct the original Nodes_of_Cells index values for comparison.

  call Initialize (Nodes_of_Cells_Index_Values, &
                   SIZE(Nodes_of_Cells_Index%Index2,1), &
                   SIZE(Nodes_of_Cells_Index%Index2,2))
  Nodes_of_Cells_Index_Values = Nodes_of_Cells_Index

  ! Check to see if the Processed Coordinates are correct.

  Success = Global_ALL(INT(Processed_Coordinates(1,:,:)) == &
            Nodes_of_Cells_Index_Values(:,:))
  call Output_Test ('BNV-AV-DV-CA-BNA Index', Success)

  Success = Global_ALL(Processed_Coordinates(2,:,:) == one)
  call Output_Test ('BNV-AV-DV-CA-BNA One', Success)

  ! A different cycle that includes an Overlapped Vector.

  Processed_Coordinates = zero
  Coordinates_Nodes_AV = Coordinates
  Coordinates_Nodes_DV = Coordinates_Nodes_AV
  Coordinates_Nodes_of_Cells_OV = Coordinates_Nodes_DV
  Coordinates_Nodes_of_Cells_CA = Coordinates_Nodes_of_Cells_OV
  Processed_Coordinates = Coordinates_Nodes_of_Cells_CA

  ! Check to see if the Processed Coordinates are correct.

  Success = Global_ALL(INT(Processed_Coordinates(1,:,:)) == &
            Nodes_of_Cells_Index_Values(:,:))
  call Output_Test ('BNV-AV-DV-OV-CA-BNA Index', Success)

  Success = Global_ALL(Processed_Coordinates(2,:,:) == one)
  call Output_Test ('BNV-AV-DV-OV-CA-BNA One', Success)

  ! Combination tests:

  !   Average test.

  call Combine_with_Average (Results_Cells_DV, Coordinates_Nodes_of_Cells_CA)
  Results_Cells_BNV = Results_Cells_DV

  Success = Global_ALL(Results_Cells_BNV(1,:) == &
            changetype(real, SUM(Nodes_of_Cells_Index_Values(:,:),2)) /  &
```

```
          changetype(real, Nodes_per_Cell))
call Output_Test ('Average index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Average one', Success)

!   Max test.

call Combine_with_MAX (Results_Cells_DV, Coordinates_Nodes_of_Cells_CA)
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          MAXVAL(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Max index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Max one', Success)

!   Min test.

call Combine_with_MIN (Results_Cells_DV, Coordinates_Nodes_of_Cells_CA)
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          MINVAL(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Min index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Min one', Success)

!   Sum test.

Results_Cells_DV = Coordinates_Nodes_of_Cells_CA  ! Sum is the default.
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          SUM(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Sum index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == four)
call Output_Test ('Sum four', Success)

! Output statements.

call Output (Coordinates_Nodes_of_Cells_CA, &
  MAX(1, Length_Total(Cell_Structure)/10), &
  MIN(Length_Total(Cell_Structure), Length_Total(Cell_Structure)/10+50) &
)

! Check state of various objects.

VERIFY(Valid_State(Coordinates_Nodes_AV),0)
VERIFY(Valid_State(Coordinates_Nodes_DV),0)
VERIFY(Valid_State(Coordinates_Nodes_of_Cells_CA),0)
VERIFY(Valid_State(Cell_Structure),0)
VERIFY(Valid_State(Node_Structure),0)
VERIFY(Valid_State(Nodes_of_Cells_Index),0)
```

```
  ! Finalize data structures and communications.

  call Finalize (Results_Cells_DV)
  call Finalize (Results_Cells_BNV)
  call Finalize (Nodes_of_Cells_Index_Values)
  call Finalize (Coordinates_Nodes_of_Cells_CA)
  call Finalize (Coordinates_Nodes_AV)
  call Finalize (Coordinates_Nodes_DV)
  call Finalize (Coordinates)
  call Finalize (Processed_Coordinates)
  call Finalize (Nodes_of_Cells_Index)
  call Finalize (Cell_Structure)
  call Finalize (Node_Structure)
  call Finalize (Comm)

end
```

Appendix E

Mathematics Module Code Listing

The main documentation of the Mathematics Module in chapter 10 on page 115 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 09/24/01
! CVS Info:   $Id: mathematics.F90,v 1.5 2005/02/26 00:23:27 hall Exp $

module Caesar_Mathematics_Module

  ! Global use associations.

  use Caesar_Data_Structures_Module
  use Caesar_Math_Utils_Module
  use Caesar_Statistics_Class

  ! Start up with everything untyped and public.
  ! Note: this module contains no private information.

  implicit none
  public

end module Caesar_Mathematics_Module
```

E.1 Math_Utils Module Code Listing

The main documentation of the Math_Utils Module in § 10.1 on page 115 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
```

```
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 01/26/05
! CVS Info:   $Id: math_utils.F90,v 1.2 2005/01/27 16:22:44 hall Exp $

module Caesar_Math_Utils_Module

  ! Global use associations.

  use Caesar_Intrinsics_Module

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Prime_Factors

  interface Extract_Factor
    module procedure Extract_Factor_Math_Utils
  end interface

  interface Prime_Factors
    module procedure Prime_Factors_Math_Utils
  end interface

contains
```

The Math_Utils Module contains the following routines which are listed in separate sections:

Prime_Factors_Math_Utils (§ E.1.1, page 472)

end module Caesar_Math_Utils_Module

E.1.1 Prime_Factors_Math_Utils Procedure

The main documentation of the Prime_Factors_Math_Utils Procedure in § 10.1.1 on page 115 contains additional explanation of this code listing.

```
  subroutine Prime_Factors_Math_Utils (Number, NFactors, Factors, Verbose)

    ! Input variables.

    type(integer), intent(in) :: Number            ! The number to be factored.
    type(logical), intent(in), optional :: Verbose ! Verbosity.

    ! Output variables.
```

```
! The number of prime factors found.
type(integer), intent(out) :: NFactors
! The vector of prime factors.
type(integer), dimension(32), intent(out) :: Factors

! Internal variables.

type(logical) :: A_Verbose        ! Actual verbosity.
type(integer) :: i                ! Loop counter.
type(integer) :: Remaining_Factor ! The current number to be factored.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Number),5)             ! Number is valid.

! Set verbosity toggle.

if (PRESENT(Verbose)) then
  A_Verbose = Verbose
else
  A_Verbose = .false.
end if

! Initializations.

Factors = 0
NFactors = 0

! Set Remaining_Factor based on sign of Number, pull out -1
! factor if negative, treat special cases of 0 and 1.

select case (Number)
case (:-1)
  Remaining_Factor = -Number
  NFactors = 1
  Factors(1) = -1
case (2:)
  Remaining_Factor = Number
case (1)
  NFactors = 1
  Factors(1) = 1
  go to 1  ! Output if requested and return.
case (0)
  NFactors = 1
  Factors(1) = 0
  go to 1  ! Output if requested and return.
end select

! Pull out all factors of 2 and 3. Note that the problem size
! is reduced with every factor that is found.
```

```
    call Extract_Factor (Remaining_Factor, Factors, NFactors, 2)
    call Extract_Factor (Remaining_Factor, Factors, NFactors, 3)

    ! Next, notice that the set of integers may be written:
    !
    !   <Integers> = {6i, 6i+1, 6i+2, 6i+3, 6i+4, 6i+5}
    !
    ! Since all factors of 2 and 3 have already been removed from
    ! Remaining_Factor, we only need to check numbers that are
    ! not divisible by 2 and 3. This rules out 6i, 6i+2, 6i+3, and
    ! 6i+4. Therefore, we only need to check numbers in the set {6i+1,
    ! 6i+5}, or equivalently, the set {6i-1, 6i+1}. The next prime
    ! number to check after 3 is 5, so we start with i=1.
    !
    ! The problem size is again reduced every time a factor is found.
    ! When the current i value becomes greater than the square root of
    ! the current problem size, we are finished. Note that the upper
    ! limit on this do-loop is unimportant -- the exit statements are
    ! always executed first.

    do i = 6, ABS(Number), 6
      call Extract_Factor (Remaining_Factor, Factors, NFactors, i-1)
      if (i - 1 > INT(sqrt(REAL(Remaining_Factor)))) exit
      call Extract_Factor (Remaining_Factor, Factors, NFactors, i+1)
      if (i + 1 > INT(sqrt(REAL(Remaining_Factor)))) exit
    end do

    ! Include the Remaining_Factor if necessary.

    if (Remaining_Factor /= 1) then
      NFactors = NFactors + 1
      Factors(NFactors) = Remaining_Factor
    end if

    ! Output.

1   if (A_Verbose) then
      write (6,*)
      write (6,100) 'Number:        ', Number
      write (6,100) 'Prime Factors: ', Factors(1:NFactors)
100   format (a,6i11,/,(15x,6i11))
    end if

    ! Verify guarantees.

    VERIFY(NFactors<=32,5)         ! Factors array is big enough.
    ! Product of factors is correct.
    VERIFY(Number==PRODUCT(Factors(1:NFactors)),5)

    return
  end subroutine Prime_Factors_Math_Utils

  ! Auxiliary procedure for Prime_Factors_Math_Utils.
```

```
  subroutine Extract_Factor_Math_Utils (Remaining_Factor, Factors, NFactors, &
                                        Divisor)

    implicit none

    ! Variables described in Prime_Factors_Math_Utils.

    integer, intent(inout) :: Factors(32), NFactors, Remaining_Factor

    ! Internal variables.

    integer, intent(in) :: Divisor  ! Divisor to be checked.
    integer :: Remainder   ! The remainder when Remaining_Factor is
                           ! divided by Divisor.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Pull out all multiples of Divisor.

    do
      Remainder = Remaining_Factor - Divisor*(Remaining_Factor/Divisor)
      if (Remainder == 0) then
        NFactors = NFactors + 1
        Factors(NFactors) = Divisor
        Remaining_Factor = Remaining_Factor/Divisor
        cycle
      end if
      exit
    end do
    return

  end subroutine Extract_Factor_Math_Utils
```

E.1.2 Math_Utils Module Unit Test Program

This lightly commented program performs a unit test on the Math_Utils Module, which is described in § 10.1 on page 115.

```
program Unit_Test

  use Caesar_Math_Utils_Module
  implicit none
  integer :: Factors(32), NFactors

  ! Testing statements.

  call Prime_Factors (         -1, NFactors, Factors, Verbose=.true.)
  call Prime_Factors (          0, NFactors, Factors, Verbose=.true.)
  call Prime_Factors (          1, NFactors, Factors, Verbose=.true.)
  call Prime_Factors (       1024, NFactors, Factors, Verbose=.true.)
  call Prime_Factors ( 1095059400, NFactors, Factors, Verbose=.true.)
  call Prime_Factors ( 1234567890, NFactors, Factors, Verbose=.true.)
```

```
  call Prime_Factors (-1073741824, NFactors, Factors, Verbose=.true.)
  call Prime_Factors (    HUGE(1), NFactors, Factors, Verbose=.true.)
  call Prime_Factors (   -HUGE(1), NFactors, Factors, Verbose=.true.)

end
```

E.2 Statistics Class Code Listing

The main documentation of the Statistics Class in § 10.2 on page 116 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 07/10/01
! CVS Info:   $Id: statistics.F90,v 3.13 2006/10/17 23:01:53 hall Exp $

module Caesar_Statistics_Class

  ! Global use associations.

  use Caesar_Data_Structures_Module

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Add_Value, Arithmetic_Mean, Average, Count, Geometric_Mean, &
            Harmonic_Mean, Maximum, Mean, Minimum, Name, Output, &
            Standard_Deviation, Sum, Total, Totally_Positive, Update_Global

  interface Initialize
    module procedure Initialize_Statistics
  end interface

  interface Finalize
    module procedure Finalize_Statistics
  end interface

  interface Valid_State
    module procedure Valid_State_Statistics
  end interface

  interface Initialized
```

```
    module procedure Initialized_Statistics
  end interface

  interface Add_Value
    module procedure Add_Value_Statistics
  end interface

  fortext([Value],[Arithmetic_Mean Count Geometric_Mean Harmonic_Mean Maximum
                   Minimum Name Standard_Deviation Sum Totally_Positive],[
    interface Value
      module procedure expand(Get_Value_Stats)
    end interface
  ])

  interface Average
    module procedure Get_Arithmetic_Mean_Stats
  end interface

  interface Mean
    module procedure Get_Arithmetic_Mean_Stats
  end interface

  interface Output
    module procedure Output_Statistics
  end interface

  interface Total
    module procedure Get_Sum_Stats
  end interface

  interface Update_Global
    module procedure Update_Global_Statistics
  end interface

  ! Public type definitions.

  public :: Statistics_type

  type Statistics_type

    ! Initialization flag.

    type(integer) :: Initialized

    ! Update status.

    type(logical) :: Global_Updated

    ! The name for this statistics object.

    type(character,80) :: Name

    ! Item count.
```

```
    type(integer) :: PE_Count, Global_Count

    ! Positivity flags.

    type(logical) :: PE_Totally_Positive
    type(logical) :: Global_Totally_Positive

    ! First order variables.

    type(real) :: PE_Arithmetic_Mean, PE_Sum
    type(real) :: PE_Geometric_Mean, PE_Log_Sum
    type(real) :: PE_Harmonic_Mean, PE_Reciprocal_Sum
    type(real) :: Global_Arithmetic_Mean, Global_Sum
    type(real) :: Global_Geometric_Mean, Global_Log_Sum
    type(real) :: Global_Harmonic_Mean, Global_Reciprocal_Sum

    ! Second order variables.

    type(real) :: PE_Squared_Sum, PE_Standard_Deviation
    type(real) :: Global_Squared_Sum, Global_Standard_Deviation

    ! Extrema.

    type(real) :: PE_Maximum, PE_Minimum
    type(real) :: Global_Maximum, Global_Minimum

  end type Statistics_type

contains
```

The Statistics Class contains the following routines which are listed in separate sections:

end module Caesar_Statistics_Class

E.2.1 Initialize_Statistics Procedure

The main documentation of the Initialize_Statistics Procedure in § 10.2.1 on page 117 contains additional explanation of this code listing.

```
subroutine Initialize_Statistics (Statistics, Name, status)

  ! Use association information.

  use Caesar_Flags_Module, only: initialized_flag
  use Caesar_Numbers_Module, only: zero

  ! Input variables.

  type(character,*), intent(in) :: Name  ! Statistics name.

  ! Output variables.

  type(Statistics_type), intent(out) :: Statistics    ! Statistics object.
  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(Status_type), dimension(25) :: allocate_status ! Allocation Status.
  type(Status_type) :: consolidated_status            ! Consolidated Status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(.not.Valid_State(Statistics),5)  ! Statistics is not valid.
  VERIFY(Valid_State(Name),5)             ! Name is valid.

  ! Allocations and initializations.

  call Initialize (allocate_status)
  call Initialize (consolidated_status)
  call Initialize (Statistics%PE_Count, allocate_status(1))
  call Initialize (Statistics%PE_Arithmetic_Mean, allocate_status(2))
  call Initialize (Statistics%PE_Sum, allocate_status(3))
  call Initialize (Statistics%PE_Geometric_Mean, allocate_status(4))
  call Initialize (Statistics%PE_Log_Sum, allocate_status(5))
  call Initialize (Statistics%PE_Harmonic_Mean, allocate_status(6))
  call Initialize (Statistics%PE_Reciprocal_Sum, allocate_status(7))
  call Initialize (Statistics%PE_Standard_Deviation, allocate_status(8))
  call Initialize (Statistics%PE_Squared_Sum, allocate_status(9))
  call Initialize (Statistics%PE_Maximum, allocate_status(10))
  call Initialize (Statistics%PE_Minimum, allocate_status(11))
  call Initialize (Statistics%PE_Totally_Positive, allocate_status(12))
  call Initialize (Statistics%Global_Count, allocate_status(13))
  call Initialize (Statistics%Global_Arithmetic_Mean, allocate_status(14))
  call Initialize (Statistics%Global_Sum, allocate_status(15))
  call Initialize (Statistics%Global_Geometric_Mean, allocate_status(16))
  call Initialize (Statistics%Global_Log_Sum, allocate_status(17))
  call Initialize (Statistics%Global_Harmonic_Mean, allocate_status(18))
  call Initialize (Statistics%Global_Reciprocal_Sum, allocate_status(19))
  call Initialize (Statistics%Global_Standard_Deviation, allocate_status(20))
  call Initialize (Statistics%Global_Squared_Sum, allocate_status(21))
```

```
call Initialize (Statistics%Global_Maximum, allocate_status(22))
call Initialize (Statistics%Global_Minimum, allocate_status(23))
call Initialize (Statistics%Global_Updated, allocate_status(24))
call Initialize (Statistics%Global_Totally_Positive, allocate_status(25))

! Set up internals.

Statistics%Name = Name

! Make sure that initial values are correct.

Statistics%PE_Count= 0
Statistics%PE_Arithmetic_Mean = zero
Statistics%PE_Sum = zero
Statistics%PE_Geometric_Mean = zero
Statistics%PE_Log_Sum = zero
Statistics%PE_Harmonic_Mean = zero
Statistics%PE_Reciprocal_Sum = zero
Statistics%PE_Standard_Deviation = zero
Statistics%PE_Squared_Sum = zero
Statistics%PE_Maximum = zero
Statistics%PE_Minimum = zero
Statistics%PE_Totally_Positive = .true.
Statistics%Global_Count= 0
Statistics%Global_Arithmetic_Mean = zero
Statistics%Global_Sum = zero
Statistics%Global_Geometric_Mean = zero
Statistics%Global_Log_Sum = zero
Statistics%Global_Harmonic_Mean = zero
Statistics%Global_Reciprocal_Sum = zero
Statistics%Global_Standard_Deviation = zero
Statistics%Global_Squared_Sum = zero
Statistics%Global_Maximum = zero
Statistics%Global_Minimum = zero
Statistics%Global_Updated = .false.
Statistics%Global_Totally_Positive = .true.

! Process status variables.

consolidated_status = allocate_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
  VERIFY(Normal(consolidated_status), 5)
end if
call Finalize (consolidated_status)
call Finalize (allocate_status)

! Set initialization flag.

Statistics%Initialized = initialized_flag

! Verify guarantees.
```

```
    VERIFY(Valid_State(Statistics),5)  ! Statistics is now valid.

    return
  end subroutine Initialize_Statistics
```

E.2.2 Finalize_Statistics Procedure

The main documentation of the Finalize_Statistics Procedure in § 10.2.2 on page 118 contains additional explanation of this code listing.

```
  subroutine Finalize_Statistics (Statistics, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Statistics to be finalized.
    type(Statistics_type), intent(inout) :: Statistics

    ! Output variables.

    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(26) :: deallocate_status ! Deallocat'n Status.
    type(Status_type) :: consolidated_status             ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Statistics),7) ! Statistics is valid.

    ! Unset initialization flag.

    Statistics%Initialized = uninitialized_flag

    ! Deallocations and finalizations.

    ! Set deallocation status.

    call Initialize (deallocate_status)
    call Initialize (consolidated_status)

    ! Finalize internals.

    call Finalize (Statistics%PE_Count, deallocate_status(1))
    call Finalize (Statistics%PE_Arithmetic_Mean, deallocate_status(2))
```

```
    call Finalize (Statistics%PE_Sum, deallocate_status(3))
    call Finalize (Statistics%PE_Geometric_Mean, deallocate_status(4))
    call Finalize (Statistics%PE_Log_Sum, deallocate_status(5))
    call Finalize (Statistics%PE_Harmonic_Mean, deallocate_status(6))
    call Finalize (Statistics%PE_Reciprocal_Sum, deallocate_status(7))
    call Finalize (Statistics%PE_Standard_Deviation, deallocate_status(8))
    call Finalize (Statistics%PE_Squared_Sum, deallocate_status(9))
    call Finalize (Statistics%PE_Maximum, deallocate_status(10))
    call Finalize (Statistics%PE_Minimum, deallocate_status(11))
    call Finalize (Statistics%PE_Totally_Positive, deallocate_status(12))
    call Finalize (Statistics%Global_Count, deallocate_status(13))
    call Finalize (Statistics%Global_Arithmetic_Mean, deallocate_status(14))
    call Finalize (Statistics%Global_Sum, deallocate_status(15))
    call Finalize (Statistics%Global_Geometric_Mean, deallocate_status(16))
    call Finalize (Statistics%Global_Log_Sum, deallocate_status(17))
    call Finalize (Statistics%Global_Harmonic_Mean, deallocate_status(18))
    call Finalize (Statistics%Global_Reciprocal_Sum, deallocate_status(19))
    call Finalize (Statistics%Global_Standard_Deviation, deallocate_status(20))
    call Finalize (Statistics%Global_Squared_Sum, deallocate_status(21))
    call Finalize (Statistics%Global_Maximum, deallocate_status(22))
    call Finalize (Statistics%Global_Minimum, deallocate_status(23))
    call Finalize (Statistics%Global_Updated, deallocate_status(24))
    call Finalize (Statistics%Global_Totally_Positive, deallocate_status(25))
    call Finalize (Statistics%Name, deallocate_status(26))

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(Statistics),7) ! Statistics is not valid.

    return
  end subroutine Finalize_Statistics
```

E.2.3 Valid_State_Statistics Procedure

The main documentation of the Valid_State_Statistics Procedure in § 10.2.3 on page 118 contains additional explanation of this code listing.

```
  function Valid_State_Statistics (Statistics) result(Valid)

    ! Use association information.
```

```
use Caesar_Numbers_Module, only: ten

! Input variables.

! Variable to be checked.
type(Statistics_type), intent(in) :: Statistics

! Output variables.

type(logical) :: Valid             ! Logical state.

! Internal variables.

type(real) :: expand_left, expand_right        ! Range expansion amounts.
type(real), dimension(2) :: Global_Mean_Range  ! Global range of the means.
type(real), dimension(2) :: Global_Range   ! Global range of the variables.
type(real), dimension(2) :: PE_Mean_Range  ! Range of the means on this PE.
type(real), dimension(2) :: PE_Range   ! Range of the variables on this PE.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Start out true.

Valid = .true.

! Check for validity of internals.

Valid = Valid .and. Initialized(Statistics)
Valid = Valid .and. Valid_State(Statistics%PE_Count)
Valid = Valid .and. Valid_State(Statistics%PE_Arithmetic_Mean)
Valid = Valid .and. Valid_State(Statistics%PE_Sum)
Valid = Valid .and. Valid_State(Statistics%PE_Geometric_Mean)
Valid = Valid .and. Valid_State(Statistics%PE_Log_Sum)
Valid = Valid .and. Valid_State(Statistics%PE_Harmonic_Mean)
Valid = Valid .and. Valid_State(Statistics%PE_Reciprocal_Sum)
Valid = Valid .and. Valid_State(Statistics%PE_Standard_Deviation)
Valid = Valid .and. Valid_State(Statistics%PE_Squared_Sum)
Valid = Valid .and. Valid_State(Statistics%PE_Maximum)
Valid = Valid .and. Valid_State(Statistics%PE_Minimum)
Valid = Valid .and. Valid_State(Statistics%PE_Totally_Positive)
Valid = Valid .and. Valid_State(Statistics%Global_Count)
Valid = Valid .and. Valid_State(Statistics%Global_Arithmetic_Mean)
Valid = Valid .and. Valid_State(Statistics%Global_Sum)
Valid = Valid .and. Valid_State(Statistics%Global_Geometric_Mean)
Valid = Valid .and. Valid_State(Statistics%Global_Log_Sum)
Valid = Valid .and. Valid_State(Statistics%Global_Harmonic_Mean)
Valid = Valid .and. Valid_State(Statistics%Global_Reciprocal_Sum)
Valid = Valid .and. Valid_State(Statistics%Global_Standard_Deviation)
Valid = Valid .and. Valid_State(Statistics%Global_Squared_Sum)
Valid = Valid .and. Valid_State(Statistics%Global_Maximum)
Valid = Valid .and. Valid_State(Statistics%Global_Minimum)
Valid = Valid .and. Valid_State(Statistics%Global_Updated)
Valid = Valid .and. Valid_State(Statistics%Global_Totally_Positive)
```

```
Valid = Valid .and. Valid_State(Statistics%Name)
if (.not.Valid) return

! Checks on the validity of Statistics.

! Range checks.

! All of the ranges have been expanded *slightly* by using the Fortran
! intrinsic SPACING. This was needed for small roundoff errors that
! were triggered when there was only a single value in the object.

! Set range expansion values.

expand_left  = ten*MAX(SPACING(Statistics%PE_Minimum), &
                       SPACING(Statistics%Global_Minimum), &
                       SPACING(Statistics%PE_Harmonic_Mean), &
                       SPACING(Statistics%Global_Harmonic_Mean) )
expand_right = ten*MAX(SPACING(Statistics%PE_Maximum), &
                       SPACING(Statistics%Global_Maximum), &
                       SPACING(Statistics%PE_Arithmetic_Mean), &
                       SPACING(Statistics%Global_Arithmetic_Mean) )

PE_Range = (/ Statistics%PE_Minimum - expand_left, &
              Statistics%PE_Maximum + expand_right /)
Valid = Valid .and. Statistics%PE_Maximum >= Statistics%PE_Minimum
Valid = Valid .and. (Statistics%PE_Arithmetic_Mean .InInterval. PE_Range)
Valid = Valid .and. (Statistics%PE_Geometric_Mean .InInterval. PE_Range)
Valid = Valid .and. (Statistics%PE_Harmonic_Mean .InInterval. PE_Range)

if (Statistics%Global_Updated) then
  Global_Range = (/ Statistics%Global_Minimum - expand_left, &
                    Statistics%Global_Maximum + expand_right /)
  Valid = Valid .and. &
          Statistics%Global_Maximum >= Statistics%Global_Minimum
  Valid = Valid .and. &
          (Statistics%PE_Maximum .InInterval. Global_Range)
  Valid = Valid .and. &
          (Statistics%PE_Minimum .InInterval. Global_Range)
  Valid = Valid .and. &
          (Statistics%Global_Arithmetic_Mean .InInterval. Global_Range)
  Valid = Valid .and. &
          (Statistics%Global_Geometric_Mean .InInterval. Global_Range)
  Valid = Valid .and. &
          (Statistics%Global_Harmonic_Mean .InInterval. Global_Range)
endif

! Mathematically, the geometric mean must be less than the arithmetic
! mean and greater than the harmonic mean.

if (Statistics%PE_Totally_Positive) then
  PE_Mean_Range = (/ Statistics%PE_Harmonic_Mean   - expand_left, &
                     Statistics%PE_Arithmetic_Mean + expand_right /)
  Valid = Valid .and. &
          (Statistics%PE_Geometric_Mean .InInterval. PE_Mean_Range)
```

```
    end if
    if (Statistics%Global_Updated .and. &
        Statistics%Global_Totally_Positive) then
      Global_Mean_Range = &
        (/ Statistics%Global_Harmonic_Mean   - expand_left, &
           Statistics%Global_Arithmetic_Mean + expand_right /)
      Valid = Valid .and. &
              (Statistics%Global_Geometric_Mean .InInterval. Global_Mean_Range)
    endif

    return
  end function Valid_State_Statistics
```

E.2.4 Initialized_Statistics Procedure

The main documentation of the Initialized_Statistics Procedure in § 10.2.4 on page 119 contains additional explanation of this code listing.

```
  function Initialized_Statistics (Statistics) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! Statistics to be checked.
    type(Statistics_type), intent(in) :: Statistics

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = Statistics%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Statistics
```

E.2.5 Add_Value_Statistics Procedure

The main documentation of the Add_Value_Statistics Procedure in § 10.2.5 on page 119 contains additional explanation of this code listing.

```
subroutine Add_Value_Statistics (Statistics, Value)

  ! Use association information.

  use Caesar_Numbers_Module, only: one, zero

  ! Input variables.

  type(real), intent(in) :: Value       ! Value to be added.

  ! Input/Output variables.

  type(Statistics_type), intent(inout) :: Statistics ! Statistics object.

  ! Internal variables.

  type(real) :: N       ! Real version of PE_Count.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Statistics),5)  ! Statistics is valid.
  VERIFY(Valid_State(Value),5)       ! Value is valid.

  ! Global stats now out of date.

  Statistics%Global_Updated = .false.

  ! Increment sums and count.

  Statistics%PE_Count = Statistics%PE_Count + 1
  Statistics%PE_Sum = Statistics%PE_Sum + Value
  if (Value > zero) then
    Statistics%PE_Log_Sum = Statistics%PE_Log_Sum + LOG(Value)
    Statistics%PE_Reciprocal_Sum = Statistics%PE_Reciprocal_Sum + one/Value
  else
    Statistics%PE_Totally_Positive = .false.
  end if
  Statistics%PE_Squared_Sum = Statistics%PE_Squared_Sum + Value**2

  ! Update statistics.

  N = changetype(real,Statistics%PE_Count)
  Statistics%PE_Arithmetic_Mean = Statistics%PE_Sum / N
  if (Statistics%PE_Totally_Positive) then
    Statistics%PE_Geometric_Mean = EXP(Statistics%PE_Log_Sum / N)
    Statistics%PE_Harmonic_Mean = N / Statistics%PE_Reciprocal_Sum
  end if
  if (Statistics%PE_Count > 1) then
    Statistics%PE_Standard_Deviation = &
      SQRT( (Statistics%PE_Squared_Sum &
      - N * Statistics%PE_Arithmetic_Mean**2) &
```

```
        / (N - one) )
    end if

    ! Update extrema.

    if (Statistics%PE_Count > 1) then
      Statistics%PE_Maximum = MAX(Statistics%PE_Maximum, Value)
      Statistics%PE_Minimum = MIN(Statistics%PE_Minimum, Value)
    else
      Statistics%PE_Maximum = Value
      Statistics%PE_Minimum = Value
    end if

    ! Verify guarantees.

    VERIFY(Valid_State(Statistics),5)        ! Statistics is valid.
    ! Means are correct.
    VERIFY(VeryClose(Statistics%PE_Arithmetic_Mean,Statistics%PE_Sum/N),5)
    if (Statistics%PE_Totally_Positive) then
      VERIFY(VeryClose(Statistics%PE_Geometric_Mean, dnl
        EXP(Statistics%PE_Log_Sum/N)),5)
      VERIFY(VeryClose(Statistics%PE_Harmonic_Mean, dnl
        N/Statistics%PE_Reciprocal_Sum),5)
    end if

    return
  end subroutine Add_Value_Statistics
```

E.2.6 Get Value Statistics Functions

The main documentation of the Get Value Statistics Functions in § 10.2.6 on page 119 contains additional explanation of this code listing.

```
  define([ACCESS_ROUTINE],[
    pushdef([VALUE], [$1])
    ifelse(
      VALUE, [Count],
        [pushdef([TYPE], [integer])],
      VALUE, [Totally_Positive],
        [pushdef([TYPE], [logical])],
      [pushdef([TYPE], [real])])
    pushdef([Get_VALUE_Stats], expand(Get_VALUE_Stats))
    pushdef([Global_VALUE], expand(Global_VALUE))
    pushdef([PE_VALUE], expand(PE_VALUE))

    function Get_VALUE_Stats (Statistics, Global) result(VALUE)

      ! Input variables.

      type(logical), optional, intent(in) :: Global    ! Global/Local toggle.

      ! Input/Output variables.
```

```
    type(Statistics_type), intent(inout) :: Statistics ! Statistics object.

    ! Output variables.

    type(TYPE) :: VALUE                    ! Statistics value to be output.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Statistics),5)      ! Statistics is valid.

    ! Set value.

    if (PRESENT(Global)) then
      if (Global) then
        call Update_Global_Statistics (Statistics)
        VALUE = Statistics%Global_VALUE
      else
        VALUE = Statistics%PE_VALUE
      end if
    else
      VALUE = Statistics%PE_VALUE
    end if

    ! Verify guarantees - none.

    return
  end function Get_VALUE_Stats

  popdef([VALUE])
  popdef([Get_VALUE_Stats])
  popdef([Global_VALUE])
  popdef([PE_VALUE])
])

fortext([Value],[Arithmetic_Mean Count Geometric_Mean Harmonic_Mean
                 Maximum Minimum Standard_Deviation Sum Totally_Positive],[
  ACCESS_ROUTINE(Value)
])

function Get_Name_Stats (Statistics) result(Name)

  ! Input variable.

  type(Statistics_type), intent(in) :: Statistics ! Statistics object.

  ! Output variable.

  type(character,80) :: Name             ! Statistics value to be output.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ! Verify requirements.

    VERIFY(Valid_State(Statistics),5)       ! Statistics is valid.

    ! Set value.

    Name = Statistics%Name

    ! Verify guarantees - none.

    return
  end function Get_Name_Stats
```

E.2.7 Output_Statistics Procedure

The main documentation of the Output_Statistics Procedure in § 10.2.7 on page 120 contains additional explanation of this code listing.

```
  subroutine Output_Statistics (Statistics, Global, Verbose, Unit)

    use Caesar_Numbers_Module, only: ten, zero

    ! Input variables.

    type(Statistics_type), intent(inout) :: Statistics ! Variable to be output.
    type(logical), intent(in), optional :: Global   ! Global flag.
    type(logical), intent(in), optional :: Verbose  ! Verbosity flag.
    type(integer), intent(in), optional :: Unit     ! Output unit.

    ! Internal variables.

    type(logical) :: A_Global                       ! Actual global flag.
    type(logical) :: A_Verbose                      ! Actual verbosity flag.
    type(integer) :: A_Unit                         ! Actual output unit.
    type(integer) :: Stat_Count                     ! Statistics count.
    type(real) :: Stat_Geometric_Mean               ! Statistics geom mean.
    type(real) :: Stat_Harmonic_Mean                ! Statistics harmonic mean.
    type(real) :: Stat_Maximum                      ! Statistics maximum.
    type(real) :: Stat_Mean                         ! Statistics mean.
    type(real) :: Stat_Minimum                      ! Statistics minimum.
    type(character,80) :: Stat_Name                 ! Statistics name.
    type(real) :: Stat_Standard_Deviation           ! Statistics std deviation.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Statistics),5)               ! Statistics is valid.

    ! Set unit number.

    if (PRESENT(Unit)) then
```

```
    A_Unit = Unit
  else
    A_Unit = 6
  end if

  ! Set global flag.

  if (PRESENT(Global)) then
    A_Global = Global
  else
    A_Global = .false.
  end if

  ! Set verbosity flag.

  if (PRESENT(Verbose)) then
    A_Verbose = Verbose
  else
    A_Verbose = .false.
  end if

  ! Do a global update if called for.

  if (A_Global) then
    call Update_Global (Statistics)
  end if

  ! These are evaluated on all PEs -- NOT inside an IO PE block -- because
  ! they contain validity checks on Statistics and thus require global
  ! communication.

  Stat_Count = Count(Statistics, A_Global)
  Stat_Name = Name(Statistics)
  Stat_Maximum = Maximum(Statistics, A_Global)
  Stat_Minimum = Minimum(Statistics, A_Global)
  Stat_Mean = Mean(Statistics, A_Global)
  Stat_Standard_Deviation = Standard_Deviation(Statistics, A_Global)
  if ((.not. A_Global .and. Statistics%PE_Totally_Positive) .or. &
      (A_Global .and. Statistics%Global_Totally_Positive)) then
    Stat_Geometric_Mean = Geometric_Mean(Statistics, A_Global)
    Stat_Harmonic_Mean = Harmonic_Mean(Statistics, A_Global)
  end if

  ! Output Statistics Info.

  if (this_is_IO_PE) then
    if (A_Verbose) then
      write (A_Unit,100) TRIM(Stat_Name), ':'
      if (A_Global) then
        write (A_Unit,101) '*Global values*'
      else
        write (A_Unit,101) '*Values for IO PE*'
      end if
      write (A_Unit,101) 'Number of values = ', Stat_Count
```

```
        write (A_Unit,102) &
          'Range            = [[', &
          Stat_Minimum, ',  ', &
          Stat_Maximum, ' ]]'
        if (Stat_Mean /= zero .and. &
            ABS(Stat_Standard_Deviation / Stat_Mean) <= ten) then
          write (A_Unit,103) &
            'Arithmetic Mean  = ', &
            Stat_Mean, ' +/- ', &
            Stat_Standard_Deviation, ' ( +/- ', &
            Stat_Standard_Deviation / &
            Stat_Mean * 100.d0, '% )'
        else
          write (A_Unit,103) &
            'Arithmetic Mean  = ', &
            Stat_Mean, ' +/- ', &
            Stat_Standard_Deviation
        end if
        if ((.not. A_Global .and. Statistics%PE_Totally_Positive) .or. &
            (A_Global .and. Statistics%Global_Totally_Positive)) then
          write (A_Unit,103) &
            'Geometric Mean   = ', Stat_Geometric_Mean
          write (A_Unit,103) &
            'Harmonic Mean    = ', Stat_Harmonic_Mean
        end if
      else
        if (Stat_Mean /= zero .and. &
            ABS(Stat_Standard_Deviation / Stat_Mean) <= ten) then
          write (A_Unit,104) &
            TRIM(Stat_Name), ' = ', &
            Stat_Mean, ' +/- ', &
            Stat_Standard_Deviation, ' ( +/- ', &
            Stat_Standard_Deviation / &
            Stat_Mean * 100.d0, '% )'
        else
          write (A_Unit,104) &
            TRIM(Stat_Name), ' = ', &
            Stat_Mean, ' +/- ', &
            Stat_Standard_Deviation
        end if
      end if
    end if

    ! Format statements.

100 format (a,a)
101 format (2x,a,i7)
102 format (2x,a,1pe14.7,a,1pe15.7,a)
103 format (2x,a,1pe15.7,:,a,1pe13.7,a,0pf8.4,a)
104 format (a,a,1pe15.7,a,1pe13.7,a,0pf7.4,a)

    ! Verify guarantees - none.

    return
```

```
  end subroutine Output_Statistics
```

E.2.8 Update_Global_Statistics Procedure

The main documentation of the Update_Global_Statistics Procedure in § 10.2.8 on page 121 contains additional explanation of this code listing.

```
  subroutine Update_Global_Statistics (Statistics)

    ! Use association information.

    use Caesar_Numbers_Module, only: one

    ! Input/Output variables.

    type(Statistics_type), intent(inout) :: Statistics ! Statistics object.

    ! Internal variables.

    type(real) :: N       ! Real version of Global_Count.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Statistics),5)  ! Statistics is valid.

    ! Check to see if the update has already been done.

    if (Statistics%Global_Updated) return

    ! Increment sums and count.

    Statistics%Global_Count = Global_Sum(Statistics%PE_Count)
    Statistics%Global_Sum = Global_Sum(Statistics%PE_Sum)
    Statistics%Global_Log_Sum = Global_Sum(Statistics%PE_Log_Sum)
    Statistics%Global_Reciprocal_Sum = &
      Global_Sum(Statistics%PE_Reciprocal_Sum)
    Statistics%Global_Squared_Sum = &
      Global_Sum(Statistics%PE_Squared_Sum)
    Statistics%Global_Totally_Positive = &
      Global_ALL(Statistics%PE_Totally_Positive)

    ! Update statistics.

    N = changetype(real,Statistics%Global_Count)
    Statistics%Global_Arithmetic_Mean = Statistics%Global_Sum / N
    if (Statistics%Global_Totally_Positive) then
      Statistics%Global_Geometric_Mean = EXP(Statistics%Global_Log_Sum / N)
      Statistics%Global_Harmonic_Mean = N / Statistics%Global_Reciprocal_Sum
    end if
    if (Statistics%Global_Count > 1) then
```

```
      Statistics%Global_Standard_Deviation = &
        SQRT( (Statistics%Global_Squared_Sum &
        - N * Statistics%Global_Arithmetic_Mean**2) &
        / (N - one) )
    end if

    ! Update extrema.

    Statistics%Global_Maximum = Global_MaxVal(Statistics%PE_Maximum)
    Statistics%Global_Minimum = Global_MinVal(Statistics%PE_Minimum)

    ! Set update flag.

    Statistics%Global_Updated = .true.

    ! Verify guarantees.

    VERIFY(Valid_State(Statistics),5)        ! Statistics is valid.
    ! Means are correct.
    VERIFY(VeryClose(Statistics%Global_Arithmetic_Mean, dnl
      Statistics%Global_Sum/N),5)
    if (Statistics%Global_Totally_Positive) then
      VERIFY(VeryClose(Statistics%Global_Geometric_Mean, dnl
        EXP(Statistics%Global_Log_Sum/N)),5)
      VERIFY(VeryClose(Statistics%Global_Harmonic_Mean, dnl
        N/Statistics%Global_Reciprocal_Sum),5)
    end if

    return
  end subroutine Update_Global_Statistics
```

E.2.9 Statistics Class Unit Test Program

This lightly commented program performs a unit test on the Statistics Class, which is described in § 10.2 on page 116.

```
program Unit_Test

  use Caesar_Data_Structures_Module
  use Caesar_Statistics_Class
  use Caesar_Numbers_Module, only: one
  implicit none

  type(Statistics_type) :: Variable_Stats
  type(Communication_type) :: Comm

  ! Initialize communications.

  call Initialize (Comm)
  call Output (Comm)
  if (this_is_IO_PE) write (6,*)
```

```
! Testing statements.

! Odd Numbers.

call Initialize (Variable_Stats, 'Odd Numbers')

call Add_Value (Variable_Stats, 1.0d0)
call Add_Value (Variable_Stats, 3.0d0)
call Add_Value (Variable_Stats, 5.0d0)
call Add_Value (Variable_Stats, 7.0d0)
call Add_Value (Variable_Stats, 9.0d0)

call Output (Variable_Stats, Global=.false., Verbose=.true.)
if (this_is_IO_PE) write (6,*)
call Output (Variable_Stats, Global=.true.,  Verbose=.true.)
if (this_is_IO_PE) write (6,*)

VERIFY(Valid_State(Variable_Stats),0)

call Finalize (Variable_Stats)

! Powers of 2.

call Initialize (Variable_Stats, 'Powers of 2')

call Add_Value (Variable_Stats, 1.0d0)
call Add_Value (Variable_Stats, 2.0d0)
call Add_Value (Variable_Stats, 4.0d0)
call Add_Value (Variable_Stats, 8.0d0)
call Add_Value (Variable_Stats, 16.0d0)
call Add_Value (Variable_Stats, 32.0d0)
call Add_Value (Variable_Stats, 64.0d0)

call Output (Variable_Stats, Global=.false., Verbose=.true.)
if (this_is_IO_PE) write (6,*)
call Output (Variable_Stats, Global=.true.,  Verbose=.true.)
if (this_is_IO_PE) write (6,*)

VERIFY(Valid_State(Variable_Stats),0)

call Finalize (Variable_Stats)

! Reciprocals.

call Initialize (Variable_Stats, 'Reciprocals')

call Add_Value (Variable_Stats, one/1.0d0)
call Add_Value (Variable_Stats, one/2.0d0)
call Add_Value (Variable_Stats, one/3.0d0)
call Add_Value (Variable_Stats, one/4.0d0)
call Add_Value (Variable_Stats, one/5.0d0)
call Add_Value (Variable_Stats, one/6.0d0)
call Add_Value (Variable_Stats, one/7.0d0)
call Add_Value (Variable_Stats, one/8.0d0)
```

```
  call Add_Value (Variable_Stats, one/9.0d0)
  call Add_Value (Variable_Stats, one/55.0d0)

  call Output (Variable_Stats, Global=.false., Verbose=.true.)
  if (this_is_IO_PE) write (6,*)
  call Output (Variable_Stats, Global=.true.,  Verbose=.true.)
  if (this_is_IO_PE) write (6,*)

  VERIFY(Valid_State(Variable_Stats),0)

  call Finalize (Variable_Stats)

  ! Negatives.

  call Initialize (Variable_Stats, 'Negatives')

  call Add_Value (Variable_Stats, 1.0d0)
  call Add_Value (Variable_Stats, -1.0d0)
  call Add_Value (Variable_Stats, 2.0d0)
  call Add_Value (Variable_Stats, -2.0d0)
  call Add_Value (Variable_Stats, 3.0d0)
  call Add_Value (Variable_Stats, -3.0d0)

  call Output (Variable_Stats, Global=.false., Verbose=.true.)
  if (this_is_IO_PE) write (6,*)
  call Output (Variable_Stats, Global=.true.,  Verbose=.true.)
  if (this_is_IO_PE) write (6,*)

  VERIFY(Valid_State(Variable_Stats),0)

  call Finalize (Variable_Stats)

  ! Parallel test.

  call Initialize (Variable_Stats, 'Parallel Test')

  call Add_Value (Variable_Stats, changetype(real, this_PE))

  call Output (Variable_Stats, Global=.false., Verbose=.true.)
  if (this_is_IO_PE) write (6,*)
  call Output (Variable_Stats, Global=.true.,  Verbose=.true.)

  VERIFY(Valid_State(Variable_Stats),0)

  call Finalize (Variable_Stats)

  ! Finalize communications.

  call Finalize (Comm)

end
```

Appendix F

Parallel_Utilities Module Code Listing

The main documentation of the Parallel_Utilities Module in chapter 11 on page 123 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D409, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 09/25/02
! CVS Info:   $Id: parallel_utilities.F90,v 1.3 2004/01/07 23:48:04 hall Exp $

module Caesar_Parallel_Utilities_Module

  ! Global use associations.

  use Caesar_Mathematics_Class
  use Caesar_Timer_Class

  ! Start up with everything untyped and public.
  ! Note: this module contains no private information.

  implicit none
  public

end module Caesar_Parallel_Utilities_Module
```

F.1 Timer Class Code Listing

The main documentation of the Timer Class in § 11.1 on page 123 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
```

```
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 07/09/01
! CVS Info:   $Id: timer.F90,v 1.20 2007/02/21 21:28:28 hall Exp $

module Caesar_Timer_Class

  ! Global use associations.

  use Caesar_Mathematics_Module
  use Caesar_Data_Structures_Module
  use Caesar_Numbers_Module, only: zero

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Arithmetic_Mean, Average, Count, Geometric_Mean, Get_CPU_Time, &
            Get_Wall_Clock_Time, Harmonic_Mean, Julian_Day, Maximum, Mean, &
            Minimum, Name, Output, Reset, Standard_Deviation, Start, Stop, &
            Sum, Total, Totally_Positive

  interface Initialize
    module procedure Initialize_Timer
  end interface

  interface Finalize
    module procedure Finalize_Timer
  end interface

  interface Valid_State
    module procedure Valid_State_Timer
  end interface

  interface Initialized
    module procedure Initialized_Timer
  end interface

  interface Average
    module procedure Get_Arithmetic_Mean_Timer
  end interface

  interface Get_CPU_Time
    module procedure Get_CPU_Time
  end interface

  interface Get_Wall_Clock_Time
    module procedure Get_Wall_Clock_Time
```

```
end interface

fortext([Value],[Arithmetic_Mean Count Geometric_Mean Harmonic_Mean
                 Maximum Minimum Name Standard_Deviation Sum
                 Totally_Positive],[
  interface Value
    module procedure expand(Get_Value_Timer)
  end interface
])

interface Mean
  module procedure Get_Arithmetic_Mean_Timer
end interface

interface Output
  module procedure Output_Timer
end interface

interface Reset
  module procedure Reset_Timer
end interface

interface Start
  module procedure Start_Timer
end interface

interface Stop
  module procedure Stop_Timer
end interface

interface Total
  module procedure Get_Sum_Timer
end interface

! Public type definitions.

public :: Time_type, Timer_type

type Time_type

  ! Starting time.

  type(real) :: Start

  ! Timing statistics.

  type(Statistics_type) :: Statistics

end type Time_type

type Timer_type

  ! Initialization status.
```

```
    type(logical,1) :: Initialized

    ! Two times to track.

    type(Time_type) :: CPU_Time, Wall_Clock_Time

    ! The name for this timer.

    type(character,80) :: Name

    ! True when the timer is active.

    type(logical) :: Running

  end type Timer_type

contains
```

The Timer Class contains the following routines which are listed in separate sections:

end module Caesar_Timer_Class

F.1.1 Initialize_Timer Procedure

The main documentation of the Initialize_Timer Procedure in § 11.1.1 on page 124 contains additional explanation of this code listing.

```
  subroutine Initialize_Timer (Timer, Name, status)

    ! Use association information.

    use Caesar_Numbers_Module, only: zero
```

```
! Input variables.

type(character,*), intent(in) :: Name  ! Timer name.

! Output variables.

type(Timer_type), intent(out) :: Timer ! Timer to be initialized.
type(Status_type), intent(out), optional :: status  ! Exit status.

! Internal variables.

type(Status_type), dimension(5) :: allocate_status  ! Allocation Status.
type(Status_type) :: consolidated_status            ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(.not.Valid_State(Timer),5)    ! Timer is not valid.
VERIFY(Valid_State(Name),5)          ! Name is valid.

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)
call Initialize (Timer%CPU_Time%Start, allocate_status(1))
call Initialize (Timer%CPU_Time%Statistics, 'CPU Time', &
                 allocate_status(2))
call Initialize (Timer%Wall_Clock_Time%Start, allocate_status(3))
call Initialize (Timer%Wall_Clock_Time%Statistics, 'Wall Clock Time', &
                 allocate_status(4))
call Initialize (Timer%Running, allocate_status(5))

! Set up internals.

Timer%Name = Name

! Make sure that initial values are correct.

Timer%CPU_Time%Start = zero
Timer%Wall_Clock_Time%Start = zero
Timer%Running = .false.

! Process status variables.

consolidated_status = allocate_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
  VERIFY(Normal(consolidated_status), 5)
end if
call Finalize (consolidated_status)
```

```
    call Finalize (allocate_status)

    ! Set initialization flag.

    call Initialize (Timer%Initialized, 0)

    ! Verify guarantees.

    VERIFY(Valid_State(Timer),5)     ! Timer is now valid.

    return
  end subroutine Initialize_Timer
```

F.1.2 Finalize_Timer Procedure

The main documentation of the Finalize_Timer Procedure in § 11.1.2 on page 125 contains additional explanation of this code listing.

```
  subroutine Finalize_Timer (Timer, status)

    ! Input/Output variable.

    ! Timer to be finalized.
    type(Timer_type), intent(inout) :: Timer

    ! Output variables.

    type(Status_type), intent(out), optional :: status   ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(6) :: deallocate_status ! Deallocation Status.
    type(Status_type) :: consolidated_status             ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Timer),7) ! Timer is valid.

    ! Unset initialization flag.

    call Finalize (Timer%Initialized)

    ! Set deallocation status.

    call Initialize (deallocate_status)
    call Initialize (consolidated_status)

    ! Finalize internals.

    call Finalize (Timer%CPU_Time%Start, deallocate_status(1))
```

```
    call Finalize (Timer%CPU_Time%Statistics, deallocate_status(2))
    call Finalize (Timer%Wall_Clock_Time%Start, deallocate_status(3))
    call Finalize (Timer%Wall_Clock_Time%Statistics, deallocate_status(4))
    call Finalize (Timer%Running, deallocate_status(5))
    call Finalize (Timer%Name, deallocate_status(6))

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(Timer),7)  ! Timer is not valid.

    return
  end subroutine Finalize_Timer
```

F.1.3 Valid_State_Timer Procedure

The main documentation of the Valid_State_Timer Procedure in § 11.1.3 on page 125 contains additional explanation of this code listing.

```
  function Valid_State_Timer (Timer) result(Valid)

    ! Input variables.

    ! Variable to be checked.
    type(Timer_type), intent(in) :: Timer

    ! Output variables.

    type(logical) :: Valid             ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for association of pointered internals.

    Valid = Valid .and. ASSOCIATED(Timer%Initialized)
    if (.not.Valid) return
```

```
    ! Check for validity of internals.

    Valid = Valid .and. Valid_State(Timer%CPU_Time%Start)
    Valid = Valid .and. Valid_State(Timer%CPU_Time%Statistics)
    Valid = Valid .and. Valid_State(Timer%Wall_Clock_Time%Start)
    Valid = Valid .and. Valid_State(Timer%Wall_Clock_Time%Statistics)
    Valid = Valid .and. Valid_State(Timer%Running)
    Valid = Valid .and. Valid_State(Timer%Name)
    if (.not.Valid) return

    ! Consistency checks.

    Valid = Valid .and. Timer%CPU_Time%Start >= zero
   ! These are giving errors -- I don't know why -- so they are commented
   ! out for now.
   ! Valid = Valid .and. (Minimum(Timer%CPU_Time%Statistics) >= zero)
    Valid = Valid .and. Timer%Wall_Clock_Time%Start >= zero
   ! Valid = Valid .and. (Minimum(Timer%Wall_Clock_Time%Statistics) >= zero)

    return
  end function Valid_State_Timer
```

F.1.4 Initialized_Timer Procedure

The main documentation of the Initialized_Timer Procedure in § 11.1.4 on page 125 contains additional explanation of this code listing.

```
  function Initialized_Timer (Timer) result(Initialized)

    ! Input variable.

    type(Timer_type), intent(in) :: Timer  ! Timer to be checked.

    ! Output variable.

    type(logical) :: Initialized           ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = ASSOCIATED(Timer%Initialized)

    ! Verify guarantees - none.

    return
  end function Initialized_Timer
```

F.1.5 Get Value Timer Functions

The main documentation of the Get Value Timer Functions in § 11.1.5 on page 126 contains additional explanation of this code listing.

```
define([ACCESS_ROUTINE],[
  pushdef([VALUE], [$1])
  ifelse(
    VALUE, [Count],
      [pushdef([TYPE], [integer])],
    VALUE, [Totally_Positive],
      [pushdef([TYPE], [logical])],
    [pushdef([TYPE], [real])])
  pushdef([Get_VALUE_Timer], expand(Get_VALUE_Timer))
  pushdef([Timer_VALUE], expand(Timer_VALUE))

  function Get_VALUE_Timer (Timer, Clock, Global, Split) result(Timer_VALUE)

    ! Input variables.

    type(character,*), optional, intent(in) :: Clock  ! Clock toggle.
    type(logical), optional, intent(in) :: Global     ! Global/Local toggle.
    type(logical), optional, intent(in) :: Split      ! Split/Overall toggle.

    ! Input/Output variables.

    type(Timer_type), intent(inout) :: Timer      ! Timer object.

    ! Output variables.

    type(TYPE) :: Timer_VALUE                     ! Timer value to be output.

    ! Internal variables.

    type(character,10) :: A_Clock                 ! Actual clock value.
    type(logical) :: A_Global                     ! Actual global flag.
    type(logical) :: A_Split                      ! Actual split flag.
    type(Statistics_type) :: PE_Stats             ! Overall PE Statistics.
    type(real) :: Total_Time_PE                   ! Total time on this PE.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Timer),5)                          ! Timer is valid.
    if (PRESENT(Clock)) then
      VERIFY(Clock == 'CPU' .or. Clock == 'Wall_Clock',5) ! Clock is valid.
    end if

    ! Set clock value.

    if (PRESENT(Clock)) then
      A_Clock = Clock
```

```
    else
      A_Clock = 'CPU'
    end if

    ! Set global flag.

    if (PRESENT(Global)) then
      A_Global = Global
    else
      A_Global = .false.
    end if

    ! Set split flag.

    if (PRESENT(Split)) then
      A_Split = Split
    else
      A_Split = .false.
    end if

    ! Set overall PE values if not reporting split values.

    if (.not.A_Split) then
      call Initialize (PE_Stats, "PE Statistics")
      if (A_Clock == 'CPU') then
        Total_Time_PE = Total(Timer%CPU_Time%Statistics)
      else if (A_Clock == 'Wall_Clock') then
        Total_Time_PE = Total(Timer%Wall_Clock_Time%Statistics)
      end if
      call Add_Value (PE_Stats, Total_Time_PE)
    end if

    ! Get VALUE from the statistics object.

    if (A_Split) then
      if (A_Clock == 'CPU') then
        Timer_VALUE = VALUE[](Timer%CPU_Time%Statistics, A_Global)
      else if (A_Clock == 'Wall_Clock') then
        Timer_VALUE = VALUE[](Timer%Wall_Clock_Time%Statistics, A_Global)
      end if
    else
      Timer_VALUE = VALUE[](PE_Stats, A_Global)
    end if

    ! Finalize PE Stats.

    if (.not.A_Split) then
      call Finalize (PE_Stats)
    end if

    ! Verify guarantees - none.

    return
  end function Get_VALUE_Timer
```

```
    popdef([VALUE])
    popdef([Get_VALUE_Timer])
    popdef([Timer_VALUE])
  ])

  fortext([Value],[Arithmetic_Mean Count Geometric_Mean Harmonic_Mean
                  Maximum Minimum Standard_Deviation Sum Totally_Positive],[
    ACCESS_ROUTINE(Value)
  ])

  function Get_Name_Timer (Timer) result(Name)

    ! Input/Output variables.

    type(Timer_type), intent(in) :: Timer ! Timer object.

    ! Output variables.

    type(character,80) :: Name              ! Timer value to be output.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Timer),5)            ! Timer is valid.

    ! Set value.

    Name = Timer%Name

    ! Verify guarantees - none.

    return
  end function Get_Name_Timer
```

F.1.6 Get_CPU_Time Procedure

The main documentation of the Get_CPU_Time Procedure in § 11.1.6 on page 127 contains additional explanation of this code listing.

```
  function Get_CPU_Time ()

    ! Output variable.

    type(real) :: Get_CPU_Time      ! CPU time counter in seconds.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Intrinsic F95 call to get cpu time info.
```

```
    call CPU_TIME (Get_CPU_Time)

    ! Verify guarantees - none.

  return
  end function Get_CPU_Time
```

F.1.7 Get_Wall_Clock_Time Procedure

The main documentation of the Get_Wall_Clock_Time Procedure in § 11.1.7 on page 127 contains additional explanation of this code listing.

```
  function Get_Wall_Clock_Time () result(Wall_Clock_Time)

    ! Use association information.

    use Caesar_Numbers_Module, only: milli

    ! Output variable.

    ! Wall Clock time counter in seconds.
    type(real) :: Wall_Clock_Time

    ! Internal variables.

    ! Date and Time array for intrinsic F95 call.
    type(integer), dimension(8) :: Date_Time
    type(integer):: Day              ! Julian Day for this date.
    type(integer) :: Seconds         ! Julian Seconds for this time.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Intrinsic F95 call to get date and time info.

    call DATE_AND_TIME (VALUES=Date_Time)

    ! Calculate Julian day for this date.

    Day = Julian_Day(Date_Time(1), Date_Time(2), Date_Time(3))

    ! Convert into a number of seconds.

    Seconds = ((Day*24 + &         ! Days.
              Date_Time(5))*60 + & ! Hours.
              Date_Time(6))*60 + & ! Minutes.
              Date_Time(7)         ! Seconds.

    ! Add in milliseconds to get the final Wall Clock time.
```

```
    Wall_Clock_Time = changetype(real,Seconds) + &
                      milli*Date_Time(8)   ! Milliseconds.

    ! Verify guarantees - none.

  return
  end function Get_Wall_Clock_Time
```

F.1.8 Julian_Day Procedure

The main documentation of the Julian_Day Procedure in § 11.1.8 on page 128 contains additional explanation of this code listing.

```
  function Julian_Day (Year, Month, Day, Calendar, Debug)

    ! Use association information.

    use Caesar_Numbers_Module, only: four

    ! Input variables.

    type(integer), intent(in) :: Year                   ! Input year.
    type(integer), intent(in) :: Month                  ! Input month.
    type(integer), intent(in) :: Day                    ! Input day.
    type(character,*), intent(in), optional :: Calendar ! Julian or
                                                        ! Gregorian calendar.
    type(logical), intent(in), optional :: Debug        ! Debug toggle.

    ! Output variables.

    type(integer) :: Julian_Day ! Julian Day.

    ! Internal variables.

    type(character,9) :: A_Calendar       ! Actual calendar.
    type(logical) :: A_Debug              ! Actual debug toggle.
    type(integer) :: Julian_Day_Constant  ! Shift for new zero date.
    type(integer) :: Julian_Year          ! Julian Year (includes zero).
    type(integer) :: Julian_Month         ! Julian Month (4-15).
    type(integer) :: Shifted_Julian_Year  ! Julian Year + 8000.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Set actual debug toggle.

    if (PRESENT(Debug)) then
      A_Debug = Debug
    else
      A_Debug = .true.
    end if

    ! Verify requirements.
```

```
if (A_Debug) then
  VERIFY(Valid_State(Year),4)         ! Year is valid.
  VERIFY(Valid_State(Month),4)        ! Month is valid.
  VERIFY(Valid_State(Day),4)          ! Day is valid.
  VERIFY(Year/=0,2)                   ! Year is non-zero.
  VERIFY(Month.InInterval.(/1,12/),2) ! Month is between 1 and 12.
  VERIFY(Day.InInterval.(/1,31/),2)   ! Day is between 1 and 31.
  ! Year is after the first Julian day occurred (1 January 4713 BC).
  VERIFY(Year >= -4713,4)
end if

! More specific requirements on Day (for higher verification levels).

if (A_Debug) then
  if (Month==4 .or. Month==6 .or. Month==9 .or. Month==11) then
    ! Day is between 1 and 30 for April, June, September and November.
    VERIFY(Day.InInterval.(/1,30/),6)
  else if (Month==2) then
    ! Day is between 1 and 29 for February.
    VERIFY(Day.InInterval.(/1,29/),6)
  end if
end if

! Set calendar to use.

if (PRESENT(Calendar)) then
  A_Calendar = Calendar
else
  A_Calendar = 'Gregorian'
end if

! Further requirements that are dependent on the calendar.

if (A_Debug) then
  if (A_Calendar == 'Gregorian') then
    ! Warn if year is before Gregorian dates were adopted in the
    ! first countries in Europe (4 October 1582 CE).
    IF_NOT_UNIT_TEST WARN_IF(Year < 1582,6)
    ! Warn if year is before Gregorian dates were adopted in
    ! England and the American Colonies (14 September 1752 CE).
    IF_NOT_UNIT_TEST WARN_IF(Year < 1752,8)
  else if (A_Calendar /= 'Julian') then
    ! Only Julian and Gregorian calendars allowed.
    VERIFY(.false.,0)
  end if
end if

! Convert to Julian (CE) Year, which includes a zero year:
!
!                           BC  AD
!     Year:     -4  -3  -2  -1  1  2  3  4
! Julian Year:  -3  -2  -1   0  1  2  3  4
```

```
if (Year > 0) then
  Julian_Year = Year
else
  Julian_Year = Year + 1
end if

! Convert to Julian Month for ease in calculating month days and
! dealing with February.
!
!     Month:      1   2  3  4  5  6  7  8   9  10  11  12
! Julian Month:  14  15  4  5  6  7  8  9  10  11  12  13
!
! For consistency, modify Julian_Year to start with March.

if (Month < 3) then
  Julian_Month = Month + 13
  Julian_Year  = Julian_Year - 1
else
  Julian_Month = Month + 1
end if

! Calculate Julian_Year shifted by 8000 years to avoid problems
! with leap years in negative years.

Shifted_Julian_Year = Julian_Year + 8000

! Adjustment constant -- this is the number of days from the start
! of the Julian Day numbering system, 1 January 4713 BC, to the new
! zero date which is used for calculations. Due to the month shifting,
! the new zero date is 30 October 2 BC.

Julian_Day_Constant = - ( &
    + 365*(-4713) &                          ! 365 days/year.
    + int(30.6001*14) &                      ! Trick to get days/month.
    + int((-4713+8000)/4) - 2000 &           ! Leap days.
    + 1 &                                    ! Day of the month.
    )

! Calculate Julian Day.

if (A_Calendar == 'Julian') then

  Julian_Day = Julian_Day_Constant &          ! Adjustment constant.
    + 365*(Julian_Year) &                     ! 365 days/year.
    + int(30.6001*Julian_Month) &             ! Trick to get days/month.
    + int(Shifted_Julian_Year/4) - 2000 &     ! Leap days.
    + Day                                     ! Day of the month.

else if (A_Calendar == 'Gregorian') then

  Julian_Day = Julian_Day_Constant &          ! Adjustment constant.
    + 365*Julian_Year &                       ! 365 days/year.
    + int(30.6001*Julian_Month) &             ! Trick to get days/month.
    + int(Shifted_Julian_Year/4)   - 2000 &   ! Leap days.
```

```
        - int(Shifted_Julian_Year/100) + 80 &     ! Gregorian leap
        + int(Shifted_Julian_Year/400) - 20 &     !   day adjustment.
        + 2 &                                     ! Difference between
                                                  !   Gregorian and Julian
                                                  !   calendars at 0 CE.
        + Day                                     ! Day of the month.

    end if

    ! Verify guarantees.

    if (A_Debug) then
      VERIFY(Valid_State(Julian_Day),4)  ! Julian_Day is valid.
      VERIFY(Julian_Day >= 0,4)          ! Julian_Day is non-negative.
    end if

    return
  end function Julian_Day
```

F.1.9 Output_Timer Procedure

The main documentation of the Output_Timer Procedure in § 11.1.9 on page 130 contains additional explanation of this code listing.

```
  subroutine Output_Timer (Timer, Global, Verbose, Unit)

    use Caesar_Numbers_Module, only: hundred, zero

    ! Input variables.

    type(Timer_type), intent(inout) :: Timer        ! Variable to be output.
    type(logical), intent(in), optional :: Global   ! Global flag.
    type(logical), intent(in), optional :: Verbose  ! Verbosity flag.
    type(integer), intent(in), optional :: Unit     ! Output unit.

    ! Internal variables.

    type(integer) :: A_Unit                     ! Actual output unit.
    type(logical) :: A_Global                   ! Actual global flag.
    type(logical) :: A_Verbose                  ! Actual verbosity flag.
    type(character,80) :: Timer_Name            ! Timer name.
    type(integer) :: Timer_Splits               ! Timer split count.
    type(real) :: Timer_CPU_Max                 ! Timer CPU maximum.
    type(real) :: Timer_CPU_Mean                ! Timer CPU mean.
    type(real) :: Timer_CPU_Min                 ! Timer CPU minimum.
    type(real) :: Timer_CPU_StDev               ! Timer CPU Standard Deviation.
    type(real) :: Timer_CPU_Split_Max           ! Timer CPU Split maximum.
    type(real) :: Timer_CPU_Split_Mean          ! Timer CPU Split mean.
    type(real) :: Timer_CPU_Split_Min           ! Timer CPU Split minimum.
    type(real) :: Timer_CPU_Split_StDev         ! Timer CPU Split Std Dev.
    type(real) :: Timer_Machine_Capacity_Usage  ! Timer machine capacity usage.
    type(real) :: Timer_Wall_Clock_Max          ! Timer Wall Clock maximum.
```

```
    type(real) :: Timer_Wall_Clock_Min          ! Timer Wall Clock minimum.
    type(real) :: Timer_Wall_Clock_Mean         ! Timer Wall Clock mean.
    type(real) :: Timer_Wall_Clock_StDev        ! Timer Wall Clock Std Dev.
    type(real) :: Timer_Wall_Clock_Split_Max    ! Timer Wall Clock Split max.
    type(real) :: Timer_Wall_Clock_Split_Min    ! Timer Wall Clock Split min.
    type(real) :: Timer_Wall_Clock_Split_Mean   ! Timer Wall Clock Split mean.
    type(real) :: Timer_Wall_Clock_Split_StDev  ! Timer Wall Clock Split StDev.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Timer),5)               ! Timer is valid.

    ! Set unit number.

    if (PRESENT(Unit)) then
      A_Unit = Unit
    else
      A_Unit = 6
    end if

    ! Set global flag.

    if (PRESENT(Global)) then
      A_Global = Global
    else
      A_Global = .false.
    end if

    ! Set verbosity flag.

    if (PRESENT(Verbose)) then
      A_Verbose = Verbose
    else
      A_Verbose = .false.
    end if

    ! Verbose output variables.

    if (A_Verbose) then

      Timer_Name = Name(Timer)
      Timer_Splits = Count(Timer, Global=A_Global, Split=.true.)

      ! CPU statistics.

      Timer_CPU_Mean = &
        Mean(Timer, 'CPU', Global=A_Global, Split=.false.)
      Timer_CPU_StDev = &
        Standard_Deviation(Timer, 'CPU', Global=A_Global, Split=.false.)
      Timer_CPU_Max = &
        Maximum(Timer, 'CPU', Global=A_Global, Split=.false.)
      Timer_CPU_Min = &
```

```
      Minimum(Timer, 'CPU', Global=A_Global, Split=.false.)

    ! Wall Clock statistics.

    Timer_Wall_Clock_Mean = &
      Mean(Timer, 'Wall_Clock', Global=A_Global, Split=.false.)
    Timer_Wall_Clock_StDev = &
      Standard_Deviation(Timer, 'Wall_Clock', Global=A_Global, Split=.false.)
    Timer_Wall_Clock_Max = &
      Maximum(Timer, 'Wall_Clock', A_Global, Split=.false.)
    Timer_Wall_Clock_Min = &
      Minimum(Timer, 'Wall_Clock', Global=A_Global, Split=.false.)

    ! Machine Capacity Usage =
    !   Average CPU Time per PE / Max Wall Clock Time

    if (Timer_Wall_Clock_Max /= zero) then
      Timer_Machine_Capacity_Usage = &
        hundred * Timer_CPU_Mean / Timer_Wall_Clock_Max
    else
      Timer_Machine_Capacity_Usage = zero
    end if

    ! CPU Split statistics.

    Timer_CPU_Split_Mean = &
      Mean(Timer, 'CPU', Global=A_Global, Split=.true.)
    Timer_CPU_Split_StDev = &
      Standard_Deviation(Timer, 'CPU', Global=A_Global, Split=.true.)
    Timer_CPU_Split_Max = &
      Maximum(Timer, 'CPU', Global=A_Global, Split=.true.)
    Timer_CPU_Split_Min = &
      Minimum(Timer, 'CPU', Global=A_Global, Split=.true.)

    ! Wall Clock Split statistics.

    Timer_Wall_Clock_Split_Mean = &
      Mean(Timer, 'Wall_Clock', Global=A_Global, Split=.true.)
    Timer_Wall_Clock_Split_StDev = &
      Standard_Deviation(Timer, 'Wall_Clock', Global=A_Global, Split=.true.)
    Timer_Wall_Clock_Split_Max = &
      Maximum(Timer, 'Wall_Clock', Global=A_Global, Split=.true.)
    Timer_Wall_Clock_Split_Min = &
      Minimum(Timer, 'Wall_Clock', Global=A_Global, Split=.true.)

  ! Non-verbose output variables.

  else

    Timer_Name = Name(Timer)
    Timer_Wall_Clock_Max = &
      Maximum(Timer, 'Wall_Clock', A_Global, Split=.false.)
    Timer_CPU_Mean = Mean(Timer, 'CPU', Global=A_Global, Split=.false.)
```

```
    end if

    ! Output Timer Info.

    if (this_is_IO_PE) then
      if (A_Verbose) then
        if (A_Global) then
          write (A_Unit,100) TRIM(Timer_Name), ': *Global values*'
          write (A_Unit,101) 'Total number of splits = ', Timer_Splits
          write (A_Unit,101) 'CPU Time / PE:'
          call Output_Stats_Timer (A_Unit, Timer_CPU_Min, &
            Timer_CPU_Max, Timer_CPU_Mean, Timer_CPU_StDev)
          write (A_Unit,101) 'Wall Clock Time / PE:'
          call Output_Stats_Timer (A_Unit, Timer_Wall_Clock_Min, &
            Timer_Wall_Clock_Max, Timer_Wall_Clock_Mean, &
            Timer_Wall_Clock_StDev)
          write (A_Unit,101) 'CPU Time / Split:'
          call Output_Stats_Timer (A_Unit, Timer_CPU_Split_Min, &
            Timer_CPU_Split_Max, Timer_CPU_Split_Mean, Timer_CPU_Split_StDev)
          write (A_Unit,101) 'Wall Clock Time / Split:'
          call Output_Stats_Timer (A_Unit, Timer_Wall_Clock_Split_Min, &
            Timer_Wall_Clock_Split_Max, Timer_Wall_Clock_Split_Mean, &
            Timer_Wall_Clock_Split_StDev)
          if (Timer_Machine_Capacity_Usage /= zero) then
            write (A_Unit,106) 'Machine Capacity Usage = ', &
              Timer_Machine_Capacity_Usage, '%'
          end if
        else
          write (A_Unit,100) TRIM(Timer_Name), ': *Values for IO PE*'
          write (A_Unit,101) 'Total number of splits = ', Timer_Splits
          write (A_Unit,104) 'CPU Time (Total)        = ', &
            Timer_CPU_Mean
          write (A_Unit,104) 'Wall Clock Time (Total) = ', &
            Timer_Wall_Clock_Mean
          write (A_Unit,101) 'CPU Time / Split:'
          call Output_Stats_Timer (A_Unit, Timer_CPU_Split_Min, &
            Timer_CPU_Split_Max, Timer_CPU_Split_Mean, Timer_CPU_Split_StDev)
          write (A_Unit,101) 'Wall Clock Time / Split:'
          call Output_Stats_Timer (A_Unit, Timer_Wall_Clock_Split_Min, &
            Timer_Wall_Clock_Split_Max, Timer_Wall_Clock_Split_Mean, &
            Timer_Wall_Clock_Split_StDev)
          if (Timer_Machine_Capacity_Usage /= zero) then
            write (A_Unit,106) 'Machine Capacity Usage = ', &
              Timer_Machine_Capacity_Usage, '%'
          end if
        end if
      else
        write (A_Unit,107) TRIM(Timer_Name), &
          ': Max Tot Wall=', Timer_Wall_Clock_Max, &
          ', Avg Tot CPU=', Timer_CPU_Mean
      end if
    end if

    ! Format statements.
```

```
100 format (a,a)
101 format (2x,a,i7)
102 format (4x,a,1pe14.7,a,1pe15.7,a)
103 format (4x,a,1pe15.7,:,a,1pe13.7,a,0pf9.4,a)
104 format (2x,a,1pe15.7)
106 format (2x,a,f11.7,a)
107 format (a,a,1pe14.7,a,1pe14.7)

    ! Verify guarantees - none.

    return
  end subroutine Output_Timer

  ! Procedure: Output_Stats_Timer
  !
  ! This supplemental private routine is only used in Output_Timer.

  subroutine Output_Stats_Timer (Unit, Min, Max, Mean, StDev)

    ! Input variables.

    type(integer), intent(in) :: Unit            ! Output unit.
    type(real), intent(in) :: Max                ! Maximum.
    type(real), intent(in) :: Mean               ! Mean.
    type(real), intent(in) :: Min                ! Minimum.
    type(real), intent(in) :: StDev              ! Standard Deviation.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none. Cannot put VERIFYs here because
    ! this routine is called from within an "if (this_is_IO_PE)"
    ! block -- no global communication allowed.

    ! Write out statistics.

    write (Unit,100) 'Range     = [[', Min, ',  ', Max, ' ]]'
    if (Mean /= zero) then
      write (Unit,101) 'Average   = ', Mean, ' +/- ', StDev, ' ( +/-', &
        StDev / Mean * 100.d0, '% )'
    else
      write (Unit,101) 'Average   = ', Mean, ' +/- ', StDev
    end if
    if (Min /= zero) then
      write (Unit,101) 'Max / Min = ', Max / Min
    else
      write (Unit,101) 'Max / Min = Infinite'
    end if

100 format (4x,a,1pe14.7,a,1pe15.7,a)
101 format (4x,a,1pe15.7,:,a,1pe13.7,a,0pf9.4,a)

    ! Verify guarantees - none.
```

```
    return
  end subroutine Output_Stats_Timer
```

F.1.10 Reset_Timer Procedure

The main documentation of the Reset_Timer Procedure in § 11.1.10 on page 130 contains additional explanation of this code listing.

```
  subroutine Reset_Timer (Timer)

    ! Input/Output variable.

    type(Timer_type), intent(inout) :: Timer  ! Timer to be reset.

    ! Internal variables.

    type(character,80) :: Name                ! Timer name.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Timer),5)  ! Timer is valid.

    ! Save name of Timer.

    Name = Timer%Name

    ! Reset Timer to initial state.

    call Finalize (Timer)
    call Initialize (Timer, Name)

    ! Verify guarantees.

    VERIFY(Valid_State(Timer),5)  ! Timer is valid.
    VERIFY(.not.Timer%Running,5)  ! Timer is not running.

    return
  end subroutine Reset_Timer
```

F.1.11 Start_Timer Procedure

The main documentation of the Start_Timer Procedure in § 11.1.11 on page 131 contains additional explanation of this code listing.

```
  subroutine Start_Timer (Timer)

    ! Input/Output variable.
```

```
    type(Timer_type), intent(inout) :: Timer  ! Timer to be started.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Timer),5)  ! Timer is valid.
    VERIFY(.not.Timer%Running,5)  ! Timer is not running.

    ! Set start times.

    Timer%Wall_Clock_Time%Start = Get_Wall_Clock_Time()
    Timer%CPU_Time%Start = Get_CPU_Time()

    ! Set status.

    Timer%Running = .true.

    ! Verify guarantees.

    VERIFY(Valid_State(Timer),5)  ! Timer is valid.
    VERIFY(Timer%Running,5)       ! Timer is running.

    return
  end subroutine Start_Timer
```

F.1.12 Stop_Timer Procedure

The main documentation of the Stop_Timer Procedure in § 11.1.12 on page 131 contains additional explanation of this code listing.

```
  subroutine Stop_Timer (Timer)

    ! Input/Output variable.

    type(Timer_type), intent(inout) :: Timer  ! Timer to be stopped.

    ! Internal variables.

    type(real) :: CPU_Time_Stop, Wall_Clock_Time_Stop

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Timer),5)  ! Timer is valid.
    VERIFY(Timer%Running,5)       ! Timer is running.

    ! Set stop times.

    CPU_Time_Stop = Get_CPU_Time()
```

```
    Wall_Clock_Time_Stop = Get_Wall_Clock_Time()

    ! Update statistics.

    call Add_Value (Timer%CPU_Time%Statistics, &
                    CPU_Time_Stop - Timer%CPU_Time%Start)
    call Add_Value (Timer%Wall_Clock_Time%Statistics, &
                    Wall_Clock_Time_Stop - Timer%Wall_Clock_Time%Start)

    ! Reset start time.

    Timer%CPU_Time%Start = zero
    Timer%Wall_Clock_Time%Start = zero

    ! Set status.

    Timer%Running = .false.

    ! Verify guarantees.

    VERIFY(Valid_State(Timer),5)  ! Timer is valid.
    VERIFY(.not.Timer%Running,5)  ! Timer is not running.

    return
  end subroutine Stop_Timer
```

F.1.13 Timer Class Unit Test Program

This lightly commented program performs a unit test on the Timer Class, which is described in § 11.1 on page 123.

```
module Unit_Test_Module
  use Caesar_Intrinsics_Module
  use Caesar_Timer_Class
  use Caesar_Communication_Class
  implicit none

contains

  subroutine Red (R)
    type(real) :: R
    type(integer) :: i
    do i = 1, 100
      R = 2+R**.781828
    end do
    return
  end subroutine Red

  subroutine Julian_Day_Output (year, month, day)
    type(integer), intent(in) :: day, month, year
    type(integer) :: julian, gregorian
```

```
    julian = Julian_Day(year,month,day,'Julian')
    gregorian = Julian_Day(year,month,day,'Gregorian')

    if (this_is_IO_PE) then
      write (6,100) 'Date: ', year, month, day, &
                    '  Julian Day:', julian, gregorian
100   format (a, i6, '/', i2, '/', i2, a, i10, i10)
    end if
    return
  end subroutine Julian_Day_Output

  subroutine Check_Interval (Hostname, Value, Min, Max)
    type(character,*) :: Hostname
    type(real) :: Value, Min, Max

    if (Value .NotInInterval. (/ Min, Max /) ) then
      if (this_is_IO_PE) then
        write (6,*) 'Hostname: ', Hostname
        write (6,*) '  **Timer not in interval**'
        write (6,*) '  Value    =  ', Value
        write (6,*) '  Interval = (', Min, ',', Max, ')'
      end if
    end if

    return
  end subroutine Check_Interval

end module Unit_Test_Module

program Unit_Test

  use Unit_Test_Module
  use Caesar_Intrinsics_Module
  use Caesar_Data_Structures_Module
  use Caesar_Timer_Class
  use Caesar_Numbers_Module, only: zero, one
  implicit none

  type(real) :: R, Timer_CPU_Mean, Timer_Wall_Clock_Max
  type(real) :: JD_avg, JD_min, JD_max, CPU_speed
  type(integer) :: day, i, Julian_Day_Number, month, month_end, year, &
                   year_end, year_start
  type(integer), dimension(12) :: days_in_month
  type(logical) :: Debug, Show_Timer_Output
  type(Timer_type) :: Blue_Loop_Timer, Julian_Day_Timer, Red_Subroutine_Timer
  type(Communication_type) :: Comm
  !--------
  ! Uncomment here and two places below to compare
  ! Wall Clock time with MPI_Wtime (parallel only).
  !type(real) :: Time_MPI, MPI_WTime
  !--------

  ! Initialize communications.
```

```
call Initialize (Comm)
call Output (Comm)
if (this_is_IO_PE) write (6,*)

! Timer output toggle. Turn off for unit tests.

Show_Timer_Output = .false.

! Initialize Timers.

call Initialize (Blue_Loop_Timer, "Blue Loop")
call Initialize (Red_Subroutine_Timer, "Red Subroutine")
call Initialize (Julian_Day_Timer, "Julian Day")

! Time some loops and subroutines.

call Start (Blue_Loop_Timer)
!--------
! Uncomment here; in declarations; and below to compare
! Wall Clock time with MPI_Wtime (parallel only).
!Time_MPI = MPI_WTime()
!--------
R = 0
do i = 1, 10000/NPEs
  call Start (Red_Subroutine_Timer)
  call Red (R)
  call Stop (Red_Subroutine_Timer)
end do
call Stop (Blue_Loop_Timer)
if (Show_Timer_Output) then
  call Output (Blue_Loop_Timer, Verbose=.true., Global=.false.)
  if (this_is_IO_PE) write (6,*) ' '
  call Output (Blue_Loop_Timer, Verbose=.true., Global=.true.)
  !--------
  ! Uncomment here; in declarations; and above to compare
  ! Wall Clock time with MPI_Wtime (parallel only).
  !if (this_is_IO_PE) write (6,*) 'MPI_WTime = ', MPI_WTime() - Time_MPI
  !--------
  if (this_is_IO_PE) write (6,*) ' '
  call Output (Red_Subroutine_Timer, Verbose=.true., Global=.false.)
  if (this_is_IO_PE) write (6,*) ' '
  call Output (Red_Subroutine_Timer, Verbose=.true., Global=.true.)
end if

! Tests on the Julian_Day procedure.

! Output some representative dates. These have all been checked.

call Start (Julian_Day_Timer)
if (this_is_IO_PE) then
  write (6,*)
  write (6,100) 'Significant dates                 Julian    Gregorian'
  write (6,100) '                                  Calendar  Calendar'
end if
```

```
call Julian_Day_Output (-4713,  1,  1)
call Julian_Day_Output ( -753,  4, 21)
call Julian_Day_Output (   -2, 10, 30)
call Julian_Day_Output (   -1,  1,  1)
call Julian_Day_Output (    1,  1,  1)
call Julian_Day_Output (  200,  2, 28)
call Julian_Day_Output (  200,  2, 29)
call Julian_Day_Output (  200,  3,  1)
call Julian_Day_Output (  300,  2, 28)
call Julian_Day_Output (  300,  2, 29)
call Julian_Day_Output (  300,  3,  1)
call Julian_Day_Output ( 1582, 10,  4)
call Julian_Day_Output ( 1582, 10, 14)
call Julian_Day_Output ( 1752,  9,  2)
call Julian_Day_Output ( 1752,  9, 13)
call Julian_Day_Output ( 1858, 11, 16)
call Julian_Day_Output ( 1968,  5, 23)
call Julian_Day_Output ( 1995, 10,  9)
call Julian_Day_Output ( 2000,  1,  1)
call Julian_Day_Output ( 2132,  8, 31)
if (this_is_IO_PE) write (6,*) ' '

! Turn off verifications inside the Julian_Day procedure for parallel
! versions of the thousands of calls in the next few tests.

if (parallel) Debug = .false.

! Julian = Gregorian Test:
!
! Julian Day numbers for Julian calendar and Gregorian calendar dates will
! generally be different. They are only the same for dates from March 1st,
! 200, to February 28, 300. Here we check to make sure they are the same
! for those dates.

days_in_month = (/ 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 /)

if (this_is_IO_PE) write (6,100) 'Julian = Gregorian test starting...'
do year = 200, 300
  do month = 1, 12
    if ((year /= 200 .or. month >= 3) .and. &
        (year /= 300 .or. month <= 2)) then
      month_end = days_in_month(month)
      if (MOD(year,4) == 0 .and. month == 2) then
        month_end = month_end + 1
      end if
      do day = 1, month_end
        if (Julian_Day(year, month, day, 'Gregorian', Debug) /= &
            Julian_Day(year, month, day, 'Julian', Debug) .and. &
            this_is_IO_PE) then
          write (6,*) '******************************************'
          write (6,*) 'Error -- Julian Day number not the same for'
          write (6,*) 'Gregorian and Julian calendars on: '
          write (6,*) ' '
          write (6,*) '  Date = ', year, month, day
```

```
          write (6,*) ' '
          write (6,*) 'and it should be.'
          write (6,*) '*******************************************'
        end if
      end do
    end if
  end do
end do
if (this_is_IO_PE) then
  write (6,100) 'Julian = Gregorian test finished.'
  write (6,*)
end if
call Stop (Julian_Day_Timer)

! Julian Calendar Sequential Test.
!
! Check to see if all Julian Days are sequential for the Julian calendar.

call Start (Julian_Day_Timer)
if (this_is_IO_PE) &
  write (6,100) 'Julian Calendar Sequential test starting...'
year_start = -4713
year_end = 2100
Julian_Day_Number = Julian_Day(year_start, 1, 1, 'Julian')
do year = year_start, year_end
  if (year /= 0) then
    do month = 1, 12
      month_end = days_in_month(month)
      if (year > 0) then
        if (MOD(year,4) == 0 .and. month == 2) then
          month_end = month_end + 1
        end if
      else
        if (MOD(year,4) == -1 .and. month == 2) then
          month_end = month_end + 1
        end if
      end if
      do day = 1, month_end
        if (Julian_Day(year, month, day, 'Julian', Debug) /= &
            Julian_Day_Number .and. this_is_IO_PE) then
          write (6,*) '**********************************************'
          write (6,*) 'Error -- Julian Day number not sequential for:'
          write (6,*) ' '
          write (6,*) '  Date = ', year, month, day
          write (6,*) ' '
          write (6,*) 'and it should be.'
          write (6,*) '**********************************************'
        end if
        Julian_Day_Number = Julian_Day_Number + 1
      end do
    end do
  end if
end do
if (this_is_IO_PE) then
```

```
    write (6,100) 'Julian Calendar Sequential test finished.'
    write (6,*)
  end if
  call Stop (Julian_Day_Timer)

  ! Gregorian Calendar Sequential Test.
  !
  ! Check to see if all Julian Days are sequential for the Gregorian calendar.

  call Start (Julian_Day_Timer)
  if (this_is_IO_PE) &
    write (6,100) 'Gregorian Calendar Sequential test starting...'
  year_start = -4713
  year_end = 2100
  Julian_Day_Number = Julian_Day(year_start, 1, 1, 'Gregorian')
  do year = year_start, year_end
    if (year /= 0) then
      do month = 1, 12
        month_end = days_in_month(month)
        if (year > 0) then
          if (MOD(year,4) == 0 .and. month == 2) then
            month_end = month_end + 1
          end if
          if (MOD(year,100) == 0 .and. MOD(year,400) /= 0 .and. &
              month == 2) then
            month_end = month_end - 1
          end if
        else
          if (MOD(year,4) == -1 .and. month == 2) then
            month_end = month_end + 1
          end if
          if (MOD(year,100) == -1 .and. MOD(year,400) /= -1 .and. &
              month == 2) then
            month_end = month_end - 1
          end if
        end if
        do day = 1, month_end
          if (Julian_Day(year, month, day, 'Gregorian', Debug) /= &
              Julian_Day_Number .and. this_is_IO_PE) then
            write (6,*) '**********************************************'
            write (6,*) 'Error -- Julian Day number not sequential for:'
            write (6,*) ' '
            write (6,*) '  Date = ', year, month, day
            write (6,*) ' '
            write (6,*) 'and it should be.'
            write (6,*) '**********************************************'
          end if
          Julian_Day_Number = Julian_Day_Number + 1
        end do
      end do
    end if
  end do
  if (this_is_IO_PE) then
    write (6,100) 'Gregorian Calendar Sequential test finished.'
```

```
    write (6,*)
  end if
  call Stop (Julian_Day_Timer)
  if (Show_Timer_Output) then
    call Output (Julian_Day_Timer, Verbose=.true., Global=.false.)
    if (this_is_IO_PE) write (6,*)
    call Output (Julian_Day_Timer, Verbose=.true., Global=.true.)
  end if

  ! Check timings for various systems.

  Timer_CPU_Mean = &
    Mean(Julian_Day_Timer, 'CPU', Global=.true., Split=.false.)
  Timer_Wall_Clock_Max = &
    Maximum(Julian_Day_Timer, 'Wall_Clock', Global=.true., Split=.false.)

  ! Turn this on for new systems to see the timings.

  if (.false.) then
    if (this_is_IO_PE) then
      write (6,*) ' Timer_CPU_Mean       = ', Timer_CPU_Mean
      write (6,*) ' Timer_Wall_Clock_Max = ', Timer_Wall_Clock_Max
    end if
  end if

  ! Galt: Xeon_Intel_Linux-2.4.2_Absoft-8.2
  !       Xeon_Intel_Linux-2.4.2_Lahey-L6.00c
  ! NPES: 2
  ! uname -a: Linux galt 2.4.2-2smp #1 SMP \
  !           Sun Apr 8 20:21:34 EDT 2001 i686 unknown
  ! f90 -V f.f90: Copyright Absoft Corporation 1994-2003; \
  !               Absoft Pro FORTRAN Version 8.2
  ! lf95 --version: Lahey/Fujitsu Fortran 95 Express Release L6.00c
  !
  !                    Lahey                 Absoft
  ! Run times:      CPU    Wall Clock    CPU    Wall Clock
  !   serial        1.32    1.32        1.47    1.47
  !   1-parallel    1.32    1.32        1.45    1.45
  !   2-parallel    1.22    1.24         .75    1.46
  !   4-parallel    1.25    2.54         .40    1.56
  !   8-parallel    1.27    5.1          .24    1.78
  !  16-parallel    1.30   10.6          .16    2.53
  !  32-parallel    1.36   22.4          .13    4.52
  !
  ! Comment: After scrutiny, I determined that the Absoft compiler is
  ! optimizing out the work on the PEs that don't need to communicate
  ! back! This is with no optimization on! Full output shows that,
  ! for example, the 32-PE CPU time is 1.46 on the IO_PE and an
  ! average of 0.08 on the other 31 PEs, resulting in an overall average
  ! of 0.13.

  if ('HOSTNAME' == 'galt') then
    if ('COMPILER' == 'Lahey') then
      if (parallel) then
```

```
      call Check_Interval ('Galt', Timer_CPU_Mean, 1.20d0, 1.50d0)
      JD_avg = 1.25 + MAX(0, NPEs-2)*0.63
      JD_min = 0.9 * JD_avg
      JD_max = 1.2 * JD_avg
      call Check_Interval ('Galt', Timer_Wall_Clock_Max, JD_min, JD_max)
    else
      call Check_Interval ('Galt', Timer_CPU_Mean, 1.25d0, 1.50d0)
      call Check_Interval ('Galt', Timer_Wall_Clock_Max, 1.25d0, 1.50d0)
    end if
  else if ('COMPILER' == 'Absoft') then
    if (parallel) then
      JD_avg = 1.2298 * NPEs**(-0.70916)
      JD_min = 0.8 * JD_avg
      JD_max = 1.2 * JD_avg
      call Check_Interval ('Galt', Timer_CPU_Mean, JD_min, JD_max)
      JD_avg = 1.3586 * EXP( 0.037611 * NPEs )
      JD_min = 0.9 * JD_avg
      JD_max = 1.3 * JD_avg
      call Check_Interval ('Galt', Timer_Wall_Clock_Max, JD_min, JD_max)
    else
      call Check_Interval ('Galt', Timer_CPU_Mean, 1.30d0, 1.60d0)
      call Check_Interval ('Galt', Timer_Wall_Clock_Max, 1.30d0, 1.60d0)
    end if
  end if

! Dagny: PentiumIII_Intel_Linux-2.4.20_Absoft-8.2
! uname -a: Linux dagny 2.4.20-emp_2420p6a0328 #1 \
!          Tue Apr 1 19:52:06 EST 2003 i686 i686 i386 GNU/Linux
! f90 -V f.f90: Copyright Absoft Corporation 1994-2003; \
!              Absoft Pro FORTRAN Version 8.2
!
! Run times:     CPU    Wall Clock
!   serial       2.33    2.33
!   1-parallel   2.33    2.33
!   2-parallel   1.20    2.41
!   4-parallel    .64    2.64
!   8-parallel    .37    3.40
!  16-parallel    .25    4.81
!  32-parallel    .19    9.16
!
! See comments on Absoft compiler under Galt above.

else if ('HOSTNAME' == 'dagny') then
  if ('COMPILER' == 'Absoft') then
    if (parallel) then
      JD_avg = 2.0016 * NPEs**(-0.73317)
      JD_min = 0.8 * JD_avg
      JD_max = 1.3 * JD_avg
      call Check_Interval ('Dagny', Timer_CPU_Mean, JD_min, JD_max)
      JD_avg = 2.1876 + 0.11487 * NPEs + 0.0032143 * NPEs**2
      JD_min = 0.9 * JD_avg
      JD_max = 1.2 * JD_avg
      call Check_Interval ('Dagny', Timer_Wall_Clock_Max, JD_min, JD_max)
    else
```

```
      call Check_Interval ('Dagny', Timer_CPU_Mean, 2.20d0, 2.50d0)
      call Check_Interval ('Dagny', Timer_Wall_Clock_Max, 2.20d0, 2.50d0)
    end if
  end if

! Kira: PentiumM_Intel_Linux-2.4.23_Absoft-8.2
! uname -a: Linux kira 2.4.23-emp_2423sw #1 \
!           Mon Dec 8 20:12:14 EST 2003 i686 i686 i386 GNU/Linux
! f90 -V f.f90: Copyright Absoft Corporation 1994-2003; \
!               Absoft Pro FORTRAN Version 8.2
!
! The PentiumM chip in kira varies its processor speed from time to time.
! Two times are given below:
!
!                   600 MHz                1600 MHz
! Run times:     CPU    Wall Clock      CPU    Wall Clock
!   serial       3.19    3.20           1.2       1.2
!   1-parallel   3.21    3.20           1.2       1.2
!   2-parallel   1.66    3.51           0.62      1.25
!   4-parallel    .90    3.82           0.33      1.37
!   8-parallel    .52    4.62           0.195     1.64
!  16-parallel    .32    6.24           0.122     2.27
!  32-parallel    .24   10.22           0.090     3.98
!
! The current CPU_speed variable can be seen via "gmake environment".
! The current speed can then be set below to allow correct timing
! results.
!
! See comments on Absoft compiler under Galt above.

else if ('HOSTNAME' == 'kira') then
  CPU_speed = 1600.d0
  if ('COMPILER' == 'Absoft') then
    if (parallel) then
      JD_avg = 1702.14 * NPEs**(-0.76068) / CPU_speed
      JD_min = 0.8 * JD_avg
      JD_max = 1.2 * JD_avg
      call Check_Interval ('Kira', Timer_CPU_Mean, JD_min, JD_max)
      JD_avg = (1767.78 + 133.542 * NPEs) / CPU_speed
      JD_min = 0.9 * JD_avg
      JD_max = 1.3 * JD_avg
      call Check_Interval ('Kira', Timer_Wall_Clock_Max, JD_min, JD_max)
    else
      JD_avg = 1920.0 / CPU_speed
      JD_min = JD_avg - 0.1
      JD_max = JD_avg + 0.1
      call Check_Interval ('Kira', Timer_CPU_Mean, JD_min, JD_max)
      JD_avg = 1950.0 / CPU_speed
      JD_min = JD_avg - 0.1
      JD_max = JD_avg + 0.1
      call Check_Interval ('Kira', Timer_Wall_Clock_Max, JD_min, JD_max)
    end if
  end if
```

```
  ! Caesar:   UltraSPARC-IIi_Sun_Solaris-5.6_Native-6.2
  ! uname -a: SunOS caesar 5.6 Generic_105181-03 sun4u sparc SUNW,Ultra-5_10
  ! f90 -V:   Sun WorkShop 6 update 2 Fortran 95 6.2 2001/05/15
  !
  ! Note that CPU Time scales with the number of PEs! This must
  ! mean that Suns return the CPU Time of the parent process. Also,
  ! some runs have given results that lead me to suspect that Suns
  ! return Wall Clock Time from the CPU_TIME call -- not sure about
  ! this.
  !
  ! Run times:     Both CPU and       Both CPU and
  !                 Wall Clock     Wall Clock (optimized)
  !   serial            7.2              4.7
  !   1-parallel        7.06             4.4
  !   2-parallel       14.35             9.9
  !   4-parallel       29.8             19.1
  !   8-parallel       58.1             41.4
  !  16-parallel      123.8             99.1
  !  32-parallel      281.3            191.0

  else if ('HOSTNAME' == 'caesar') then
    if (parallel) then
      JD_avg = 1.631 + 3.261 * NPEs + 0.2697 * NPEs**2 - 0.005832 * NPEs**3
      ! Non-optimized version:
      !JD_avg = 1.011 + 6.626 * NPEs + 0.06659 * NPEs**2
      JD_min = 0.8 * JD_avg
      JD_max = 1.1 * JD_avg
      call Check_Interval ('Caesar', Timer_CPU_Mean, JD_min, JD_max)
      call Check_Interval ('Caesar', Timer_Wall_Clock_Max, JD_min, JD_max)
    else
      JD_min = 4.4d0
      JD_max = 5.0d0
      ! Non-optimized version:
      ! JD_min = 6.9d0
      ! JD_max = 7.5d0
      call Check_Interval ('Caesar', Timer_CPU_Mean, JD_min, JD_max)
      call Check_Interval ('Caesar', Timer_Wall_Clock_Max, JD_min, JD_max)
    end if
  end if

  ! Check state of Timer.

  VERIFY(Valid_State(Blue_Loop_Timer),0)
  VERIFY(Valid_State(Red_Subroutine_Timer),0)
  VERIFY(Valid_State(Julian_Day_Timer),0)

  ! Finalize Timers.

  call Finalize (Blue_Loop_Timer)
  call Finalize (Red_Subroutine_Timer)
  call Finalize (Julian_Day_Timer)

  ! Finalize communications.
```

```
  call Finalize (Comm)

  ! Format statements.

100 format (a)

end
```

Appendix G

Linear_Algebra Module Code Listing

The main documentation of the Linear_Algebra Module in chapter 12 on page 133 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 09/16/03
! CVS Info:   $Id: linear_algebra.F90,v 1.6 2004/06/15 22:57:50 hall Exp $

module Caesar_Linear_Algebra_Module

  ! Global use associations.

  use Caesar_Data_Structures_Module
  use Caesar_Mathematic_Vector_Class
  use Caesar_ELL_Matrix_Class
  use Caesar_Solver_Class
  !use Caesar_Matrix_Class
  !use Caesar_Full_Matrix_Class

  ! Start up with everything untyped and public.
  ! Note: this module contains no private information.

  implicit none
  public

end module Caesar_Linear_Algebra_Module
```

G.1 Mathematic_Vector Class Code Listing

The main documentation of the Mathematic_Vector Class in § 12.1 on page 133 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 12/15/02
! CVS Info:   $Id: mathematic_vector.F90,v 1.36 2009/09/11 21:44:06 hall Exp $

module Caesar_Mathematic_Vector_Class

  ! Global use associations.

  use Caesar_Data_Structures_Module

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Assignment(=), Operator(.DotProduct.), Operator(.Orthogonal.)
  public :: Add_Values, Average, Duplicate, First_PE, Get_Values, &
            Infinity_Norm, Last_PE, Length_PE, Length_Total, Locus, Maximum, &
            Mean, Minimum, Name, Norm, One_Norm, Output, P_Norm, &
            Set_Not_Up_to_Date, Set_Value, Set_Values, Sum, Total, Two_Norm, &
            Update_DV

  interface Initialize
    module procedure Initialize_Mathematic_Vector
  end interface

  interface Finalize
    module procedure Finalize_Mathematic_Vector
  end interface

  interface Valid_State
    module procedure Valid_State_Mathematic_Vector
  end interface

  interface Initialized
    module procedure Initialized_Mathematic_Vector
  end interface

  interface Assignment(=)
    module procedure Get_Values_Mathematic_Vector
    module procedure Set_Values_Mathematic_Vector_A
    module procedure Set_Value_Mathematic_Vector_A
  end interface

  interface OPERATOR (.DotProduct.)
```

```
    module procedure DotProduct_Mathematic_Vector
  end interface

  interface OPERATOR (.Orthogonal.)
    module procedure Orthogonal_Mathematic_Vector
  end interface

  interface Add_Values
    module procedure Add_Values_Mathematic_Vector_0
    module procedure Add_Values_Mathematic_Vector_1
    module procedure Add_Values_Mathematic_Vector_A
  end interface

  fortext([Value],[Average First_PE Infinity_Norm Last_PE Length_PE
                   Length_Total Locus Maximum Minimum Name One_Norm
                   P_Norm Sum Two_Norm],[
    interface Value
      module procedure expand(Get_Value_MV)
    end interface
  ])

  interface DotProduct
    module procedure DotProduct_Mathematic_Vector
  end interface

  interface Duplicate
    module procedure Duplicate_Mathematic_Vector
  end interface

  interface Get_Values
    module procedure Get_Values_Mathematic_Vector
  end interface

  interface Mean
    module procedure Get_Average_MV
  end interface

  interface Norm
    module procedure Get_Two_Norm_MV
  end interface

  interface Orthogonal
    module procedure Orthogonal_Mathematic_Vector
  end interface

  interface Output
    module procedure Output_Mathematic_Vector
  end interface

  interface Set_Not_Up_to_Date
    module procedure Set_Not_Up_to_Date_MV
  end interface

  interface Set_Value
```

```
    module procedure Set_Value_Mathematic_Vector_A
  end interface

  interface Set_Values
    module procedure Set_Values_Mathematic_Vector_0
    module procedure Set_Values_Mathematic_Vector_1
    module procedure Set_Values_Mathematic_Vector_A
  end interface

  interface Total
    module procedure Get_Sum_MV
  end interface

  interface Update_DV
    module procedure Update_DV_Mathematic_Vector
  end interface

  ! Public variable (must precede type defs where it is used).

  type(integer), parameter :: Number_of_OVs_in_an_MV=4
  public :: Number_of_OVs_in_an_MV

  ! Public type definitions.

  public :: Mathematic_Vector_type

  type Mathematic_Vector_type

    ! Initialization flag.

    type(integer) :: Initialized

    ! The name for this variable (especially useful in a vector of
    ! Mathematic Vectors).

    type(character,name_length) :: Name

    ! Basic data structure.

    type(Base_Structure_type), pointer :: Structure

    ! Values for the Mathematic Vector.

    type(real,1) :: Values

    ! MV dimensionality is unity.

    type(integer) :: Dimensionality=1

    ! Distributed and Overlapped Vectors that are used for matvecs.

    type(Distributed_Vector_type) :: DV
    type(Overlapped_Vector_type), dimension(Number_of_OVs_in_an_MV) :: OV
```

```
    ! Pointers to the Data_Index for each Overlapped Vector -- cannot
    ! have an array of pointers in F90 (Arrgh). Note that there are four
    ! of these because the current value of Number_of_OVs_in_an_MV is 4.

    type(Data_Index_type), pointer :: Index1, Index2, Index3, Index4

    ! Number used to match with a similar number for a matrix during a
    ! MatVec. This would be better done by using a pointered Data_Index
    ! in the matrix and checking association status, but that won't work
    ! currently in F90 -- pointered internals should only be used to point
    ! to things initialized elsewhere, and cannot be initialized themselves
    ! because they contain no internals. So, the current solution is somewhat
    ! of a kludge.

    type(integer), dimension(Number_of_OVs_in_an_MV) :: Index_Match_Number

    ! Norm variables and "updated?" toggles.

    type(real) :: Average, Infinity_Norm, Maximum, Minimum, One_Norm, P_Norm, &
                  Sum, Two_Norm
    type(integer) :: P_Norm_Exponent
    type(logical) :: Average_is_Updated, Infinity_Norm_is_Updated, &
                     Maximum_is_Updated, Minimum_is_Updated, &
                     One_Norm_is_Updated, DV_is_Updated, P_Norm_is_Updated, &
                     Sum_is_Updated, Two_Norm_is_Updated

  end type Mathematic_Vector_type

contains
```

The Mathematic_Vector Class contains the following routines which are listed in separate sections:

Update_DV_Mathematic_Vector (§ G.1.14, page 561)

end module Caesar_Mathematic_Vector_Class

G.1.1 Initialize_Mathematic_Vector Procedure

The main documentation of the Initialize_Mathematic_Vector Procedure in § 12.1.1 on page 135 contains additional explanation of this code listing.

```
  subroutine Initialize_Mathematic_Vector (MV, Structure, Name, status)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Use association information.

    use Caesar_Numbers_Module, only: zero

    ! Input variables.

    ! Base_Structure.
    type(Base_Structure_type), intent(in), target :: Structure
    type(character,*), intent(in), optional :: Name     ! Variable name.

    ! Output variables.

    ! Mathematic_Vector to be initialized.
    type(Mathematic_Vector_type), intent(out) :: MV
    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(19) :: allocate_status ! Allocation Status.
    type(Status_type) :: consolidated_status           ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Structure),5)  ! Structure is valid.

    ! Set up structure pointer.

    MV%Structure => Structure

    ! Allocations and initializations.

    call Initialize (allocate_status)
    call Initialize (consolidated_status)
    call Initialize (MV%Values, Length_PE(Structure), allocate_status(1))

    ! Set up internals.
```

```
    if (PRESENT(Name)) MV%Name = Name
    call Initialize (MV%Average,                  allocate_status(2))
    call Initialize (MV%Average_is_Updated,       allocate_status(3))
    call Initialize (MV%Infinity_Norm,            allocate_status(4))
    call Initialize (MV%Infinity_Norm_is_Updated, allocate_status(5))
    call Initialize (MV%Maximum,                  allocate_status(6))
    call Initialize (MV%Maximum_is_Updated,       allocate_status(7))
    call Initialize (MV%Minimum,                  allocate_status(8))
    call Initialize (MV%Minimum_is_Updated,       allocate_status(9))
    call Initialize (MV%One_Norm,                 allocate_status(10))
    call Initialize (MV%One_Norm_is_Updated,      allocate_status(11))
    call Initialize (MV%DV_is_Updated,            allocate_status(12))
    call Initialize (MV%P_Norm,                   allocate_status(13))
    call Initialize (MV%P_Norm_Exponent,          allocate_status(14))
    call Initialize (MV%P_Norm_is_Updated,        allocate_status(15))
    call Initialize (MV%Sum,                      allocate_status(16))
    call Initialize (MV%Sum_is_Updated,           allocate_status(17))
    call Initialize (MV%Two_Norm,                 allocate_status(18))
    call Initialize (MV%Two_Norm_is_Updated,      allocate_status(19))

    ! Make sure that initial values are correct.

    MV%Average = zero
    MV%Average_is_Updated = .false.
    MV%Infinity_Norm = zero
    MV%Infinity_Norm_is_Updated = .false.
    MV%Maximum = zero
    MV%Maximum_is_Updated = .false.
    MV%Minimum = zero
    MV%Minimum_is_Updated = .false.
    MV%One_Norm = zero
    MV%One_Norm_is_Updated = .false.
    MV%DV_is_Updated = .false.
    MV%P_Norm = zero
    MV%P_Norm_Exponent = 1
    MV%P_Norm_is_Updated = .false.
    MV%Sum = zero
    MV%Sum_is_Updated = .false.
    MV%Two_Norm = zero
    MV%Two_Norm_is_Updated = .false.

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)
```

```
    ! Set initialization flag.

    MV%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(MV),5)     ! Mathematic_Vector is now valid.

    return
  end subroutine Initialize_Mathematic_Vector
```

G.1.2 Duplicate_Mathematic_Vector Procedure

The main documentation of the Duplicate_Mathematic_Vector Procedure in § 12.1.2 on page 136 contains additional explanation of this code listing.

```
  subroutine Duplicate_Mathematic_Vector (MV_duplicate, MV_source, status)

    ! Input variables.

    type(Mathematic_Vector_type), intent(in) :: MV_source

    ! Output variables.

    ! Mathematic_Vector to be initialized.
    type(Mathematic_Vector_type), intent(out) :: MV_duplicate
    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(Status_type) :: A_status  ! Actual status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV_source),5)         ! MV_source is valid.

    ! Allocate MV_duplicate to be like MV_source.

    call Initialize (A_status)
    call Initialize (MV_duplicate, MV_source%Structure, MV_source%Name, &
                     A_status)

    ! Process status variable.

    if (PRESENT(status)) then
      WARN_IF(Error(A_status), 5)
      status = A_status
    else
      VERIFY(Normal(A_status), 5)
    end if
```

```
    call Finalize (A_status)

    ! Copy internals from source mathematical vector.

    MV_duplicate%Average                  = MV_source%Average
    MV_duplicate%Average_is_Updated       = MV_source%Average_is_Updated
    MV_duplicate%Infinity_Norm            = MV_source%Infinity_Norm
    MV_duplicate%Infinity_Norm_is_Updated = MV_source%Infinity_Norm_is_Updated
    MV_duplicate%Maximum                  = MV_source%Maximum
    MV_duplicate%Maximum_is_Updated       = MV_source%Maximum_is_Updated
    MV_duplicate%Minimum                  = MV_source%Minimum
    MV_duplicate%Minimum_is_Updated       = MV_source%Minimum_is_Updated
    MV_duplicate%One_Norm                 = MV_source%One_Norm
    MV_duplicate%One_Norm_is_Updated      = MV_source%One_Norm_is_Updated
    MV_duplicate%P_Norm                   = MV_source%P_Norm
    MV_duplicate%P_Norm_Exponent          = MV_source%P_Norm_Exponent
    MV_duplicate%P_Norm_is_Updated        = MV_source%P_Norm_is_Updated
    MV_duplicate%Sum                      = MV_source%Sum
    MV_duplicate%Sum_is_Updated           = MV_source%Sum_is_Updated
    MV_duplicate%Two_Norm                 = MV_source%Two_Norm
    MV_duplicate%Two_Norm_is_Updated      = MV_source%Two_Norm_is_Updated
    MV_duplicate%Values                   = MV_source%Values

    ! Verify guarantees.

    VERIFY(Valid_State(MV_duplicate),5)     ! Mathematic_Vector is now valid.

    return
  end subroutine Duplicate_Mathematic_Vector
```

G.1.3 Finalize_Mathematic_Vector Procedure

The main documentation of the Finalize_Mathematic_Vector Procedure in § 12.1.3 on page 136 contains additional explanation of this code listing.

```
  subroutine Finalize_Mathematic_Vector (MV, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Mathematic_Vector to be finalized.
    type(Mathematic_Vector_type), intent(inout) :: MV

    ! Output variables.

    type(Status_type), intent(out), optional :: status   ! Exit status.

    ! Internal variables.
```

```
type(integer) :: i                                   ! Loop variable.
! Deallocation Status.
type(Status_type), dimension(Number_of_OVs_in_an_MV+21) :: &
  deallocate_status
type(Status_type) :: consolidated_status             ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(MV),7)                ! Mathematic_Vector is valid.
VERIFY(Number_of_OVs_in_an_MV==4,5)      ! 4 is assumed in this procedure.

! Unset initialization flag.

MV%Initialized = uninitialized_flag

! Deallocations and finalizations.

! Set deallocation status.

call Initialize (deallocate_status)
call Initialize (consolidated_status)

! Finalize internals.

NULLIFY(MV%Structure)
NULLIFY(MV%Index1)
NULLIFY(MV%Index2)
NULLIFY(MV%Index3)
NULLIFY(MV%Index4)
call Finalize (MV%Name,                        deallocate_status( 1))
call Finalize (MV%Average,                     deallocate_status( 2))
call Finalize (MV%Average_is_Updated,          deallocate_status( 3))
call Finalize (MV%Infinity_Norm,               deallocate_status( 4))
call Finalize (MV%Infinity_Norm_is_Updated,    deallocate_status( 5))
call Finalize (MV%Maximum,                     deallocate_status( 6))
call Finalize (MV%Maximum_is_Updated,          deallocate_status( 7))
call Finalize (MV%Minimum,                     deallocate_status( 8))
call Finalize (MV%Minimum_is_Updated,          deallocate_status( 9))
call Finalize (MV%One_Norm,                    deallocate_status(10))
call Finalize (MV%One_Norm_is_Updated,         deallocate_status(11))
call Finalize (MV%DV_is_Updated,               deallocate_status(12))
call Finalize (MV%P_Norm,                      deallocate_status(13))
call Finalize (MV%P_Norm_Exponent,             deallocate_status(14))
call Finalize (MV%P_Norm_is_Updated,           deallocate_status(15))
call Finalize (MV%Sum,                         deallocate_status(16))
call Finalize (MV%Sum_is_Updated,              deallocate_status(17))
call Finalize (MV%Two_Norm,                    deallocate_status(18))
call Finalize (MV%Two_Norm_is_Updated,         deallocate_status(19))
call Finalize (MV%Values,                      deallocate_status(20))
do i = 1, Number_of_OVs_in_an_MV
  if (Initialized(MV%OV(i))) then
    call Finalize (MV%OV(i), deallocate_status(i+21))
```

```
      end if
    end do
    if (Initialized(MV%DV)) call Finalize (MV%DV, deallocate_status(21))

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(MV),7)          ! Mathematic_Vector is not valid.

    return
  end subroutine Finalize_Mathematic_Vector
```

G.1.4 Valid_State_Mathematic_Vector Procedure

The main documentation of the Valid_State_Mathematic_Vector Procedure in § 12.1.4 on page 137 contains additional explanation of this code listing.

```
  function Valid_State_Mathematic_Vector (MV) result(Valid)

    ! Use association information.

    use Caesar_Numbers_Module, only: zero, ten

    ! Input variables.

    ! Variable to be checked.
    type(Mathematic_Vector_type), intent(in) :: MV

    ! Output variables.

    type(logical) :: Valid             ! Logical state.

    ! Internal variables.

    type(integer) :: i                 ! Loop variable.
    type(real) :: N                    ! Total length of the MV.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.
```

```
Valid = .true.

! Check for association of pointered internals.

Valid = Valid .and. ASSOCIATED(MV%Structure)
if (.not.Valid) return

! Check for validity of internals.

Valid = Valid .and. Initialized(MV)
Valid = Valid .and. Valid_State(MV%Average)
Valid = Valid .and. Valid_State(MV%Average_is_Updated)
if (Initialized(MV%DV)) then
  Valid = Valid .and. Valid_State(MV%DV)
end if
Valid = Valid .and. Valid_State(MV%Infinity_Norm)
Valid = Valid .and. Valid_State(MV%Infinity_Norm_is_Updated)
Valid = Valid .and. Valid_State(MV%Maximum)
Valid = Valid .and. Valid_State(MV%Maximum_is_Updated)
Valid = Valid .and. Valid_State(MV%Minimum)
Valid = Valid .and. Valid_State(MV%Minimum_is_Updated)
Valid = Valid .and. Valid_State(MV%Name)
Valid = Valid .and. Valid_State(MV%One_Norm)
Valid = Valid .and. Valid_State(MV%One_Norm_is_Updated)
do i = 1, Number_of_OVs_in_an_MV
  if (Initialized(MV%OV(i))) then
    Valid = Valid .and. Valid_State(MV%OV(i))
  end if
end do
Valid = Valid .and. Valid_State(MV%DV_is_Updated)
Valid = Valid .and. Valid_State(MV%P_Norm)
Valid = Valid .and. Valid_State(MV%P_Norm_Exponent)
Valid = Valid .and. Valid_State(MV%P_Norm_is_Updated)
Valid = Valid .and. Valid_State(MV%Structure)
Valid = Valid .and. Valid_State(MV%Sum)
Valid = Valid .and. Valid_State(MV%Sum_is_Updated)
Valid = Valid .and. Valid_State(MV%Two_Norm)
Valid = Valid .and. Valid_State(MV%Two_Norm_is_Updated)
if (.not.Valid) return

! Checks on the validity of Mathematic_Vector.

Valid = Valid .and. MV%Infinity_Norm >= zero
Valid = Valid .and. MV%One_Norm >= zero
Valid = Valid .and. MV%P_Norm >= zero
Valid = Valid .and. MV%Two_Norm >= zero
if (MV%One_Norm_is_updated .and. MV%Sum_is_updated) then
  Valid = Valid .and. MV%One_Norm >= MV%Sum
end if
if (MV%Average_is_updated .and. MV%Sum_is_updated) then
  Valid = Valid .and. &
          VeryClose(MV%Average, MV%Sum/Length_Total(MV%Structure))
end if
if (MV%Maximum_is_updated .and. MV%Minimum_is_updated) then
```

```
      Valid = Valid .and. MV%Maximum >= MV%Minimum
    end if
    if (.not.Valid) return

    ! Mathematic relationship checks.
    ! SPACING calls are used to avoid problems with round-off.

    N = changetype(real, Length_Total(MV%Structure))
    if (MV%One_Norm_is_updated .and. MV%Two_Norm_is_updated) then
      Valid = Valid .and. &
              MV%One_Norm >= MV%Two_Norm - ten * SPACING(MV%Two_Norm)
      Valid = Valid .and. &
              MV%Two_Norm * SQRT(N) >= MV%One_Norm - ten * SPACING(MV%One_Norm)
    end if
    if (MV%Infinity_Norm_is_updated .and. MV%Two_Norm_is_updated) then
      Valid = Valid .and. &
              MV%Two_Norm >= MV%Infinity_Norm - ten * SPACING(MV%Infinity_Norm)
      Valid = Valid .and. &
              MV%Infinity_Norm * SQRT(N) >= MV%Two_Norm &
                                              - ten * SPACING(MV%Two_Norm)
    end if
    if (MV%Infinity_Norm_is_updated .and. MV%One_Norm_is_updated) then
      Valid = Valid .and. &
              MV%One_Norm >= MV%Infinity_Norm - ten * SPACING(MV%Infinity_Norm)
      Valid = Valid .and. &
              MV%Infinity_Norm * N >= MV%One_Norm - ten * SPACING(MV%One_Norm)
    end if

    return
  end function Valid_State_Mathematic_Vector
```

G.1.5 Initialized_Mathematic_Vector Procedure

The main documentation of the Initialized_Mathematic_Vector Procedure in § 12.1.5 on page 137 contains additional explanation of this code listing.

```
  function Initialized_Mathematic_Vector (MV) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! Mathematic_Vector to be checked.
    type(Mathematic_Vector_type), intent(in) :: MV

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = MV%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Mathematic_Vector
```

G.1.6 Add_Values_Mathematic_Vector Procedure

The main documentation of the Add_Values_Mathematic_Vector Procedure in § 12.1.6 on page 137 contains additional explanation of this code listing.

```
  subroutine Add_Values_Mathematic_Vector_0 (MV, Value, Row, Global)

    ! Note: This procedure is very similar to Set_Values_Mathematic_Vector_0.

    ! Input variable.

    type(real), intent(in) :: Value               ! Value scalar.
    type(integer), intent(in) :: Row              ! Row integer scalar.
    type(logical), intent(in), optional :: Global ! Global/local index toggle.

    ! Input/Output variable.

    ! Variable to be incremented.
    type(Mathematic_Vector_type), intent(inout) :: MV

    ! Internal variables.

    type(logical) :: A_Global                     ! Actual global/local toggle.
    type(integer) :: shift                        ! Index shift.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV),5)                         ! MV is valid.
    VERIFY(Valid_State(Value),5)                      ! Value is valid.
    VERIFY(Valid_State(Row),5)                        ! Row is valid.

    ! Global/Local toggle.

    if (PRESENT(Global)) then
      A_Global = Global
    else
      A_Global = .true.
    end if
```

```
    if (A_Global) then
      shift = -First_PE(MV%Structure) + 1
    else
      shift = 0
    end if

    ! Another requirement check -- require that Row be in the correct range.

    VERIFY(Row + shift .InInterval. (/1, Length_PE(MV%Structure)/),5)

    ! Add the value.

    if (Row /= 0) then
      MV%Values(Row + shift) = MV%Values(Row + shift) + Value
    end if

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (MV)

    ! Verify guarantees.

    VERIFY(Valid_State(MV),5)                            ! MV is still valid.

    return
  end subroutine Add_Values_Mathematic_Vector_0

  subroutine Add_Values_Mathematic_Vector_1 (MV, Values, Rows, Global)

    ! Note: This procedure is very similar to Set_Values_Mathematic_Vector_1.

    ! Input variable.

    type(real,1,np), intent(in) :: Values          ! Values bare naked vector.
    type(integer,1,np), intent(in) :: Rows         ! Rows integer vector.
    type(logical), intent(in), optional :: Global  ! Global/local index toggle.

    ! Input/Output variable.

    ! Variable to be incremented.
    type(Mathematic_Vector_type), intent(inout) :: MV

    ! Internal variables.

    type(logical) :: A_Global                     ! Actual global/local toggle.
    type(integer) :: i                            ! Loop parameter.
    ifelse(m4_eval(DEBUG_LEVEL >= 9), 1, [
      type(integer) :: j                          ! Loop parameter.
    ])
    type(integer) :: shift                        ! Index shift.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.
```

```
    VERIFY(Valid_State(MV),5)                          ! MV is valid.
    VERIFY(Valid_State_NP(Values),5)                   ! Values is valid.
    VERIFY(Valid_State_NP(Rows),5)                     ! Rows is valid.
    VERIFY(SIZE(Values) <= Length_PE(MV%Structure),5)  ! Values size check.
    VERIFY(SIZE(Rows) <= Length_PE(MV%Structure),5)    ! Rows size check.
    VERIFY(SIZE(Rows) == SIZE(Values),5)               ! Values/Rows check.

    ! Global/Local toggle.

    if (PRESENT(Global)) then
      A_Global = Global
    else
      A_Global = .true.
    end if
    if (A_Global) then
      shift = -First_PE(MV%Structure) + 1
    else
      shift = 0
    end if

    ! More requirement checks -- require that Rows entries are in the
    ! correct range.

    VERIFY(Rows + shift >= 1,5)
    VERIFY(Rows + shift <= Length_PE(MV%Structure),5)

    ! Add the values.

    do i = 1, SIZE(Values)
      if (Rows(i) /= 0) then
        MV%Values(Rows(i) + shift) = MV%Values(Rows(i) + shift) + Values(i)
      end if
    end do
    ! Make sure no rows are added twice.
    VERIFY((/(((Rows(i)/=Rows(j)), j=i+1,SIZE(Values)), i=1,SIZE(Values))/),9)

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (MV)

    ! Verify guarantees.

    VERIFY(Valid_State(MV),5)                            ! MV is still valid.

    return
  end subroutine Add_Values_Mathematic_Vector_1

  subroutine Add_Values_Mathematic_Vector_A (MV, Values)

    ! Note: This procedure is very similar to Set_Values_Mathematic_Vector_A.

    ! Input variable.
```

```
    type(real,1,np), intent(in) :: Values         ! Values bare naked vector.

    ! Input/Output variable.

    ! Variable to be incremented.
    type(Mathematic_Vector_type), intent(inout) :: MV

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV),5)                          ! MV is valid.
    VERIFY(Valid_State_NP(Values),5)                   ! Values is valid.
    VERIFY(SIZE(Values) == Length_PE(MV%Structure),5)  ! Values size check.

    ! Add the values.

    MV%Values = MV%Values + Values

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (MV)

    ! Verify guarantees.

    VERIFY(Valid_State(MV),5)                            ! MV is still valid.

    return
  end subroutine Add_Values_Mathematic_Vector_A
```

G.1.7 DotProduct_Mathematic_Vector Procedure

The main documentation of the DotProduct_Mathematic_Vector Procedure in § 12.1.7 on page 138 contains additional explanation of this code listing.

```
  function DotProduct_Mathematic_Vector (MV1, MV2) result(DotProduct)

    ! Input variables.

    ! Mathematic Vectors to be dotted.
    type(Mathematic_Vector_type), intent(in) :: MV1, MV2

    ! Output variable.

    type(real) :: DotProduct            ! Result of dot product.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV1),5)          ! MV1 is valid.
    VERIFY(Valid_State(MV2),5)          ! MV2 is valid.
```

```
    ! Calculate the global dot product.

    DotProduct = Global_Dot_Product(MV1%Values, MV2%Values)

    ! Verify guarantees.

    ! Cauchy-Schwartz inequality must be satisfied (only checked if the
    ! Two Norms have already been calculated). The inequality is:
    !
    !   |DotProduct| <= ||MV1||_2 * ||MV2||_2
    !
    ! but is coded in a more convoluted form below to handle close to equal
    ! comparisons correctly.

    if (MV1%Two_Norm_is_Updated .and. MV2%Two_Norm_is_Updated) then
      VERIFY(MV1%Two_Norm*MV2%Two_Norm - ABS(DotProduct) >= dnl
             -EPSILON(DotProduct)*ABS(DotProduct),5)
    end if

    return
  end function DotProduct_Mathematic_Vector
```

G.1.8 Get Value Mathematic_Vector Functions

The main documentation of the Get Value Mathematic_Vector Functions in § 12.1.8 on page 138 contains additional explanation of this code listing.

```
  define([REAL_ACCESS_ROUTINE],[
    pushdef([VALUE], [$1])
    pushdef([Get_REAL_VALUE_MV], expand(Get_VALUE_MV))
    pushdef([VALUE_is_Updated], expand(VALUE_is_Updated))

    function Get_REAL_VALUE_MV (MV) result(VALUE)

      ! Input/Output variables.

      ! Mathematic_Vector object.
      type(Mathematic_Vector_type), intent(inout) :: MV

      ! Output variables.

      type(real) :: VALUE            ! Mathematic_Vector value to be output.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(MV),5)      ! Mathematic_Vector is valid.

      ! Set value.
```

```
    if (MV%VALUE_is_Updated) then
      VALUE = MV%VALUE
    else
      ifelse(
        VALUE, [Average],
          [VALUE = Global_Sum(MV%Values)/Length_Total(MV%Structure)],
        VALUE, [Infinity_Norm],
          [VALUE = Global_MaxVal(ABS(MV%Values))],
        VALUE, [Maximum],
          [VALUE = Global_MaxVal(MV%Values)],
        VALUE, [Minimum],
          [VALUE = Global_MinVal(MV%Values)],
        VALUE, [One_Norm],
          [VALUE = Global_Sum(ABS(MV%Values))],
        VALUE, [Sum],
          [VALUE = Global_Sum(MV%Values)],
        VALUE, [Two_Norm],
          [VALUE = SQRT(Global_Sum(MV%Values**2))],
        [])
      MV%VALUE_is_Updated = .true.
      MV%VALUE = VALUE
    end if

    ! Verify guarantees - none.

    return
  end function Get_REAL_VALUE_MV

  popdef([VALUE])
  popdef([Get_REAL_VALUE_MV])
  popdef([VALUE_is_Updated])
])

fortext([Value],
        [Average Infinity_Norm Maximum Minimum One_Norm Sum Two_Norm],[
  REAL_ACCESS_ROUTINE(Value)
])

define([INTEGER_ACCESS_ROUTINE],[
  pushdef([VALUE], [$1])
  pushdef([VALUE_Result], expand(VALUE_Result))
  pushdef([Get_INTEGER_VALUE_MV], expand(Get_VALUE_MV))

  function Get_INTEGER_VALUE_MV (MV) result(VALUE_Result)

    ! Input/Output variables.

    ! Mathematic_Vector object.
    type(Mathematic_Vector_type), intent(inout) :: MV

    ! Output variables.

    type(integer) :: VALUE_Result   ! Mathematic_Vector value to be output.
```

```
    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV),5)       ! Mathematic_Vector is valid.

    ! Set value.

    ifelse(
      VALUE, [First_PE],
        [VALUE_Result = First_PE(MV%Structure)],
      VALUE, [Last_PE],
        [VALUE_Result = Last_PE(MV%Structure)],
      VALUE, [Length_PE],
        [VALUE_Result = Length_PE(MV%Structure)],
      VALUE, [Length_Total],
        [VALUE_Result = Length_Total(MV%Structure)],
      [])

    ! Verify guarantees - none.

    return
  end function Get_INTEGER_VALUE_MV

  popdef([VALUE])
  popdef([VALUE_Result])
  popdef([Get_INTEGER_VALUE_MV])
])

fortext([Value],
        [First_PE Last_PE Length_PE Length_Total],[
  INTEGER_ACCESS_ROUTINE(Value)
])

function Get_Locus_MV (MV) result(Locus_MV)

  ! Input variables.

  ! Mathematic_Vector object.
  type(Mathematic_Vector_type), intent(in) :: MV

  ! Output variables.

  ! Mathematic_Vector value to be output.
  type(character,name_length) :: Locus_MV

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(MV),5)       ! Mathematic_Vector is valid.

  ! Set value.
```

```
    Locus_MV = Locus(MV%Structure)

    ! Verify guarantees - none.

    return
  end function Get_Locus_MV

  function Get_Name_MV (MV) result(Name)

    ! Input variables.

    ! Mathematic_Vector object.
    type(Mathematic_Vector_type), intent(in) :: MV

    ! Output variables.

    ! Mathematic_Vector value to be output.
    type(character,name_length) :: Name

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV),5)       ! Mathematic_Vector is valid.

    ! Set value.

    Name = MV%Name

    ! Verify guarantees - none.

    return
  end function Get_Name_MV

  function Get_P_Norm_MV (MV, P) result(P_Norm)

    ! Use association information.

    use Caesar_Numbers_Module, only: one

    ! Input variables.

    type(integer), optional, intent(in) :: P    ! P_Norm exponent.

    ! Input/Output variables.

    ! Mathematic_Vector object.
    type(Mathematic_Vector_type), intent(inout) :: MV

    ! Output variables.

    type(real) :: P_Norm            ! Mathematic_Vector value to be output.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ! Verify requirements.

    VERIFY(Valid_State(MV),5)       ! Mathematic_Vector is valid.

    ! Set value.

    if (MV%P_Norm_is_Updated .and. P==MV%P_Norm_Exponent) then
      P_Norm = MV%P_Norm
    else
      P_Norm = (Global_Sum(ABS(MV%Values)**P))**(one/P)
      MV%P_Norm_is_Updated = .true.
      MV%P_Norm = P_Norm
      MV%P_Norm_Exponent = P
    end if

    ! Verify guarantees - none.

    return
  end function Get_P_Norm_MV
```

G.1.9 Get_Values_Mathematic_Vector Procedure

The main documentation of the Get_Values_Mathematic_Vector Procedure in § 12.1.9 on page 139 contains additional explanation of this code listing.

```
  subroutine Get_Values_Mathematic_Vector (Values, MV)

    ! Input variable.

    type(Mathematic_Vector_type), intent(in) :: MV ! Variable to be queried.

    ! Input/Output variable.

    type(real,1,np), intent(inout) :: Values        ! Values bare naked vector.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV),5)                          ! MV is valid.
    VERIFY(Valid_State_NP(Values),5)                   ! Values is valid.
    VERIFY(SIZE(Values) == Length_PE(MV%Structure),5)  ! Values size check.

    ! Get the values.

    Values = MV%Values

    ! Verify guarantees.

    VERIFY(Valid_State(MV),5)                          ! MV is still valid.
```

```
    return
  end subroutine Get_Values_Mathematic_Vector
```

G.1.10 Orthogonal_Mathematic_Vector Procedure

The main documentation of the Orthogonal_Mathematic_Vector Procedure in § 12.1.10 on page 140 contains additional explanation of this code listing.

```
  function Orthogonal_Mathematic_Vector (MV1, MV2) result(Orthogonal)

    ! Use association information.

    use Caesar_Numbers_Module, only: zero

    ! Input variables.

    ! Mathematic Vectors to be dotted.
    type(Mathematic_Vector_type), intent(in) :: MV1, MV2

    ! Output variable.

    type(logical) :: Orthogonal         ! Result of dot product.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV1),5)          ! MV1 is valid.
    VERIFY(Valid_State(MV2),5)          ! MV2 is valid.

    ! Calculate whether the two vectors are orthogonal.

    Orthogonal = Dot_Product(MV1%Values, MV2%Values) .eq. zero

    ! Verify guarantees -- none.

    return
  end function Orthogonal_Mathematic_Vector
```

G.1.11 Output_Mathematic_Vector Procedure

The main documentation of the Output_Mathematic_Vector Procedure in § 12.1.11 on page 140 contains additional explanation of this code listing.

```
  subroutine Output_Mathematic_Vector (MV, First, Last, Unit, Indent)

    ! Input variables.

    ! Variable to be output.
```

```
type(Mathematic_Vector_type), intent(inout) :: MV
type(integer), intent(in), optional :: First    ! Extents of value data
type(integer), intent(in), optional :: Last     !   to be output.
type(integer), intent(in), optional :: Unit     ! Output unit.
type(integer), optional :: Indent               ! Indentation.

! Internal variables.

type(integer) :: Buffer_Loc                      ! Buffer location.
type(integer) :: Buffer_Size                     ! Output buffer size.
type(integer) :: Buffer_Skip                     ! Buffer increment.
type(integer) :: i_global, i_local               ! Loop counters.
type(integer) :: A_First                         ! Actual first value.
type(integer) :: A_Last                          ! Actual last value.
type(integer) :: A_Unit                          ! Actual output unit.
type(integer) :: A_Indent                        ! Actual indentation.
type(character,80) :: Blanks                     ! A line of blanks.
type(character,80) :: MV_Name                    ! Name of the MV.
type(character,80) :: Output_1                   ! Output buffer.
type(character,80,1) :: Output_Buffer            ! Output buffer vector.
type(real) :: MV_Average, MV_Infinity_Norm, &    ! Get Value variables.
              MV_Maximum, MV_Minimum, &
              MV_One_Norm, MV_P_Norm, MV_Sum, &
              MV_Two_Norm

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(MV),5)      ! MV is valid.

! Set unit number.

if (PRESENT(Unit)) then
  A_Unit = Unit
else
  A_Unit = 6
end if

! Set indentation.

if (PRESENT(Indent)) then
  A_Indent = Indent
else
  A_Indent = 0
end if
Blanks = ' '

! These are evaluated on all PEs -- NOT inside an IO PE block -- because
! they contain validity checks on MV and thus require global communication.

MV_Name = Name(MV)
MV_Average = Average(MV)
MV_Infinity_Norm = Infinity_Norm(MV)
```

```
    MV_Maximum = Maximum(MV)
    MV_Minimum = Minimum(MV)
    MV_One_Norm = One_Norm(MV)
    MV_P_Norm = P_Norm(MV, MV%P_Norm_Exponent)
    MV_Sum = Sum(MV)
    MV_Two_Norm = Two_Norm(MV)

    ! Output Identification Info.

    if (this_is_IO_PE) then
      write (A_Unit,100) Blanks(1:A_Indent), 'Mathematic Vector Information:'
      write (A_Unit,101) Blanks(1:A_Indent+2), 'Name                = ', &
                         TRIM(MV_Name)
      write (A_Unit,102) Blanks(1:A_Indent+2), 'Initialized         = ', &
                         Initialized(MV)
      write (A_Unit,103) Blanks(1:A_Indent+2), 'Dimensionality      =', &
                         MV%Dimensionality
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Average             =', &
                         MV_Average
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Maximum             =', &
                         MV_Maximum
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Minimum             =', &
                         MV_Minimum
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Sum                 =', &
                         MV_Sum
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Infinity_Norm       =', &
                         MV_Infinity_Norm
      write (A_Unit,104) Blanks(1:A_Indent+2), 'One_Norm            =', &
                         MV_One_Norm
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Two_Norm            =', &
                         MV_Two_Norm
      write (A_Unit,103) Blanks(1:A_Indent+2), 'P_Norm_Exponent     =', &
                         MV%P_Norm_Exponent
      write (A_Unit,104) Blanks(1:A_Indent+2), 'P_Norm              =', &
                         MV_P_Norm
    end if

    ! Output internal structure info.

    call Output (MV%Structure, A_Unit, 'Base', A_Indent+2)

    ! Output internal values.

    if (this_is_IO_PE) then
      write (A_Unit,100) Blanks(1:A_Indent), '  Internal Values:'
    end if

    ! Set up local limits in terms of global limits.

    if (PRESENT(First)) then
      A_First = First
    else
      A_First = 1
    end if
```

```
    if (PRESENT(Last)) then
      A_Last = Last
    else
      A_Last = Length_Total(MV%Structure)
    end if
    A_First = MAX(A_First, First_PE(MV%Structure))
    A_Last = MIN(A_Last, Last_PE(MV%Structure))

    ! Output the values.

    Buffer_Size = MAX(0, (A_Last - A_First + 1))
    call Initialize (Output_Buffer, Buffer_Size)
    if (Buffer_Size /= 0) then
      Buffer_Skip = 1
      Buffer_Loc = 1
      do i_global = A_First, A_Last
        i_local = i_global - First_PE(MV%Structure) + 1
        write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),105) &
          'PE:', this_PE, ', Values(', i_global, ') =', &
          MV%Values(i_local)
        Buffer_Loc = Buffer_Loc + Buffer_Skip
      end do
    end if

    ! Add indentation.

    do Buffer_loc = 1, Buffer_Size
      Output_1 = Output_Buffer(Buffer_loc)
      Output_Buffer(Buffer_loc) = Blanks(1:A_Indent) // Output_1
    end do

    call Parallel_Write (Output_Buffer, A_Unit)
    call Finalize (Output_Buffer)

    ! Format statements. With these formats, this should work up to
    ! (10^6 - 1) PEs.

100 format (/, 2a, /)
101 format (3a)
102 format (2a, l2)
103 format (2a, i12, :, 3(',', i12, :), a)
104 format (2a, 1p, e13.5e3)
105 format (2x, a, i5, a, i11, a, 1p, e13.5e3, :, &
            2(',', e13.5e3, :), ',', /, &
            (36x, e13.5e3, :, 2(',', e13.5e3, :), ','))

    ! Verify guarantees - none.

    return
  end subroutine Output_Mathematic_Vector
```

G.1.12 Set_Not_Up_to_Date_Mathematic_Vector Procedure

The main documentation of the Set_Not_Up_to_Date_Mathematic_Vector Procedure in § 12.1.12 on page 140 contains additional explanation of this code listing.

```
subroutine Set_Not_Up_to_Date_MV (MV)

  ! Input/Output variable.

  type(Mathematic_Vector_type), intent(inout) :: MV ! Variable to be set.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(MV),5)                          ! MV is valid.

  ! Unset the updated? variables.

  MV%Average_is_Updated       = .false.
  MV%Infinity_Norm_is_Updated = .false.
  MV%Maximum_is_Updated       = .false.
  MV%Minimum_is_Updated       = .false.
  MV%One_Norm_is_Updated      = .false.
  MV%DV_is_Updated            = .false.
  MV%P_Norm_is_Updated        = .false.
  MV%Sum_is_Updated           = .false.
  MV%Two_Norm_is_Updated      = .false.

  ! Verify guarantees.

  VERIFY(Valid_State(MV),5)                            ! MV is still valid.

  return
end subroutine Set_Not_Up_to_Date_MV
```

G.1.13 Set_Values_Mathematic_Vector Procedure

The main documentation of the Set_Values_Mathematic_Vector Procedure in § 12.1.13 on page 141 contains additional explanation of this code listing.

```
subroutine Set_Values_Mathematic_Vector_0 (MV, Value, Row, Global)

  ! Note: This procedure is very similar to Add_Values_Mathematic_Vector_0.

  ! Input variable.

  type(real), intent(in) :: Value               ! Value scalar.
  type(integer), intent(in) :: Row              ! Row integer scalar.
  type(logical), intent(in), optional :: Global ! Global/local index toggle.
```

```
    ! Input/Output variable.

    type(Mathematic_Vector_type), intent(inout) :: MV ! Variable to be set.

    ! Internal variables.

    type(logical) :: A_Global                    ! Actual global/local toggle.
    type(integer) :: shift                       ! Index shift.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV),5)                        ! MV is valid.
    VERIFY(Valid_State(Value),5)                     ! Value is valid.
    VERIFY(Valid_State(Row),5)                       ! Row is valid.

    ! Global/Local toggle.

    if (PRESENT(Global)) then
      A_Global = Global
    else
      A_Global = .true.
    end if
    if (A_Global) then
      shift = -First_PE(MV%Structure) + 1
    else
      shift = 0
    end if

    ! Another requirement check -- require that Row be in the correct range.

    VERIFY(Row + shift .InInterval. (/1, Length_PE(MV%Structure)/),5)

    ! Set the value.

    if (Row /= 0) then
      MV%Values(Row + shift) = Value
    end if

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (MV)

    ! Verify guarantees.

    VERIFY(Valid_State(MV),5)                           ! MV is still valid.

    return
  end subroutine Set_Values_Mathematic_Vector_0

  subroutine Set_Values_Mathematic_Vector_1 (MV, Values, Rows, Global)

    ! Note: This procedure is very similar to Add_Values_Mathematic_Vector_1.
```

```
! Input variable.

type(real,1,np), intent(in) :: Values          ! Values bare naked vector.
type(integer,1,np), intent(in) :: Rows         ! Rows integer vector.
type(logical), intent(in), optional :: Global  ! Global/local index toggle.

! Input/Output variable.

type(Mathematic_Vector_type), intent(inout) :: MV ! Variable to be set.

! Internal variables.

type(logical) :: A_Global                     ! Actual global/local toggle.
type(integer) :: i                            ! Loop parameter.
ifelse(m4_eval(DEBUG_LEVEL >= 9), 1, [
  type(integer) :: j                          ! Loop parameter.
])
type(integer) :: shift                        ! Index shift.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(MV),5)                          ! MV is valid.
VERIFY(Valid_State_NP(Values),5)                   ! Values is valid.
VERIFY(Valid_State_NP(Rows),5)                     ! Rows is valid.
VERIFY(SIZE(Values) <= Length_PE(MV%Structure),5)  ! Values size check.
VERIFY(SIZE(Rows) <= Length_PE(MV%Structure),5)    ! Rows size check.
VERIFY(SIZE(Rows) == SIZE(Values),5)               ! Values/Rows check.

! Global/Local toggle.

if (PRESENT(Global)) then
  A_Global = Global
else
  A_Global = .true.
end if
if (A_Global) then
  shift = -First_PE(MV%Structure) + 1
else
  shift = 0
end if

! More requirement checks -- require that Rows entries are in the
! correct range.

VERIFY(Rows + shift >= 1,5)
VERIFY(Rows + shift <= Length_PE(MV%Structure),5)

! Set the values.

do i = 1, SIZE(Values)
  if (Rows(i) /= 0) then
```

```
      MV%Values(Rows(i) + shift) = Values(i)
    end if
  end do
  ! Make sure no rows are set twice.
  VERIFY((/(((Rows(i)/=Rows(j)), j=i+1,SIZE(Values)), i=1,SIZE(Values))/),9)

  ! Unset the updated? variables.

  call Set_Not_Up_to_Date (MV)

  ! Verify guarantees.

  VERIFY(Valid_State(MV),5)                            ! MV is still valid.

  return
end subroutine Set_Values_Mathematic_Vector_1

subroutine Set_Values_Mathematic_Vector_A (MV, Values)

  ! Note: This procedure is very similar to Add_Values_Mathematic_Vector_A.

  ! Input variable.

  type(real,1,np), intent(in) :: Values         ! Values bare naked vector.

  ! Input/Output variable.

  type(Mathematic_Vector_type), intent(inout) :: MV ! Variable to be set.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(MV),5)                          ! MV is valid.
  VERIFY(Valid_State_NP(Values),5)                   ! Values is valid.
  VERIFY(SIZE(Values) == Length_PE(MV%Structure),5)  ! Values size check.

  ! Set the values.

  MV%Values = Values

  ! Unset the updated? variables.

  call Set_Not_Up_to_Date (MV)

  ! Verify guarantees.

  VERIFY(Valid_State(MV),5)                            ! MV is still valid.

  return
end subroutine Set_Values_Mathematic_Vector_A

subroutine Set_Value_Mathematic_Vector_A (MV, Value)
```

```
    ! Input variable.

    type(real), intent(in) :: Value                   ! Value scalar.

    ! Input/Output variable.

    type(Mathematic_Vector_type), intent(inout) :: MV ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV),5)                          ! MV is valid.
    VERIFY(Valid_State(Value),5)                       ! Value is valid.

    ! Set all the values to the input scalar.

    MV%Values = Value

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (MV)

    ! Verify guarantees.

    VERIFY(Valid_State(MV),5)                          ! MV is still valid.

    return
  end subroutine Set_Value_Mathematic_Vector_A
```

G.1.14 Update_DV_Mathematic_Vector Procedure

The main documentation of the Update_DV_Mathematic_Vector Procedure in § 12.1.14 on page 141 contains additional explanation of this code listing.

```
  subroutine Update_DV_Mathematic_Vector (MV)

    ! Input/Output variable.

    type(Mathematic_Vector_type), intent(inout) :: MV ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(MV),5)                          ! MV is valid.

    ! Check to see if the DV needs to be updated.

    if (.not. MV%DV_is_Updated) then

      if (.not.Initialized(MV%DV)) then
```

```
      call Initialize (MV%DV, MV%Structure, MV%Dimensionality, &
                       MV%Name)
    end if
    MV%DV = MV%Values

  end if

  ! Set the updated? variable.

  MV%DV_is_Updated = .true.

  ! Verify guarantees.

  VERIFY(Valid_State(MV),5)                          ! MV is still valid.

  return
end subroutine Update_DV_Mathematic_Vector
```

G.1.15 Mathematic_Vector Class Unit Test Program

This lightly commented program performs a unit test on the Mathematic_Vector Class, which is described in § 12.1 on page 133.

```
program Unit_Test

  use Caesar_Data_Structures_Module
  use Caesar_Mathematic_Vector_Class
  use Caesar_Numbers_Module, only: one, two, three, six
  implicit none

  type(Communication_type) :: Comm
  type(Base_Structure_type) :: Row_Structure
  type(Status_type) :: status
  type(character,name_length) :: Locus_Name, MV_Name
  type(integer,1) :: Row_Length_Vector, Rows
  type(integer) :: shift
  type(real,1) :: Values
  type(real) :: A_Norm2, B_Norm2, DotP, NRows_real, PNorm
  type(integer) :: I, NRows
  type(Mathematic_Vector_type) :: A_MV, B_MV

  ! Initializations.

  call Initialize (Comm)
  call Output (Comm)
  call Initialize (status)
  call Initialize (Locus_Name)
  Locus_Name = 'Rows'
  call Initialize (MV_Name)
  MV_Name = 'Variables'
  call Initialize (Row_Length_Vector, NPEs)
  Row_Length_Vector = (/ (i**2, i = 1, NPEs) /)
```

```
! Local temp vectors.

call Initialize (Rows, Row_Length_Vector(this_PE))
call Initialize (Values, Row_Length_Vector(this_PE))

! Initialize base structure and mathematic vectors.

call Initialize (Row_Structure, Row_Length_Vector, Locus_Name, status)
call Initialize (A_MV, Row_Structure, MV_Name, status)
call Duplicate (B_MV, A_MV, status)
NRows = Length_Total(Row_Structure)

! Testing statements.

! Note that there can be no procedure calls inside a loop which
! is executed a different number of times on each PE, because
! global communication is done for verifications within a procedure.

shift = - First_PE(Row_Structure) + 1
do i = First_PE(Row_Structure), Last_PE(Row_Structure)
  ! Values(i) = global i
  Values(i + shift) = changetype(real, i)
  ! Rows(i) = global i, but reversed in order on each PE
  Rows(i + shift) = Last_PE(Row_Structure) + First_PE(Row_Structure) - i
end do

! Set A_MV via operator overloading.

A_MV = Values

! Set B_MV via explicit call (vector of values) with half of Values,
! then use the Add_Values procedure to add the other half.
! Note that B_MV values are in reverse order on each PE.

Values = Values/2.d0
call Set_Values (B_MV, Values, Rows)
call Add_Values (B_MV, Values, Rows)

! Output the MVs.

PNorm = P_Norm(A_MV, 4)
call Output (A_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Output (B_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check some things that are known because Values(i) = i :

if (Average(A_MV) /= (Maximum(A_MV)+one)/two) then
  if (this_is_IO_PE) then
    write (6,*) 'Error -- Average or Maximum is incorrect.'
  end if
end if
if (Maximum(A_MV) /= Infinity_Norm(A_MV)) then
  if (this_is_IO_PE) then
```

```
      write (6,*) 'Error -- Maximum or Infinity_Norm is incorrect.'
    end if
  end if
  if (Minimum(A_MV) /= one) then
    if (this_is_IO_PE) then
      write (6,*) 'Error -- Minimum is incorrect.'
    end if
  end if
  if (Sum(A_MV) /= One_Norm(A_MV)) then
    if (this_is_IO_PE) then
      write (6,*) 'Error -- Sum or One_Norm is incorrect.'
    end if
  end if

  ! Tweak one value of A_MV and re-output.

  call Set_Values (A_MV, 10.d0, Last_PE(Row_Structure))
  call Output (A_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Set A_MV = B_MV (values are reversed on each PE).

  Values = B_MV
  A_MV = Values

  ! Orthogonality test (note A_MV = B_MV from above).

  if (A_MV .Orthogonal. B_MV) then
    if (this_is_IO_PE) then
      write (6,*) 'Error -- Vectors are orthogonal.'
    end if
  end if

  ! Dot product test.
  ! This makes use of the fact that
  !
  !  --
  !  \     2     1     3      2
  !  /    k   =  - (2 n  + 3 n  + n)
  !  --          6
  ! k=1,n

  A_MV = Values             ! Make sure A_MV = B_MV. Note that value order is
  B_MV = Values             ! reversed on each PE, but that won't affect the
                            ! dot product.
  A_Norm2 = Two_Norm(A_MV)  ! These are calculated here so that the
  B_Norm2 = Two_Norm(B_MV)  ! Cauchy-Schwartz inequality check will be
                            ! done in the dot product.
  DotP = A_MV .DotProduct. B_MV
  NRows_real = changetype(real,NRows)
  if (this_is_IO_PE) then
    write (6,100) 'Dot product of A_MV and B_MV = ', DotP
100 format (/,a,f15.1)
    if (.not. VeryClose(DotP, &
        (two * NRows_real**3 + three * NRows_real**2 + NRows_real)/six)) then
```

```
      write (6,*) 'Error -- Dot product is incorrect.'
    end if
  end if

  ! Also need
  ! 1: call Add (MV1, MV2, MV3) or call Add_to_MV (MV1, MV2)
  ! 2: MV1 = MV2
  ! 3: call Scale (MV, scalar)
  ! 4: MV1 == MV2

  ! Check state of the mathematic vectors.

  VERIFY(Valid_State(A_MV),0)
  VERIFY(Valid_State(B_MV),0)

  ! Finalize mathematic vectors and communications.

  call Finalize (A_MV)
  call Finalize (B_MV)
  call Finalize (Comm)

end
```

G.2 ELL_Matrix Class Code Listing

The main documentation of the ELL_Matrix Class in § 12.2 on page 142 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 12/15/02
! CVS Info:   $Id: ell_matrix.F90,v 1.40 2009/09/11 21:44:06 hall Exp $

module Caesar_ELL_Matrix_Class

  ! Global use associations.

  use Caesar_Data_Structures_Module
  use Caesar_Mathematic_Vector_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.
```

```
public :: Initialize, Finalize, Valid_State, Initialized
public :: Assignment(=)
public :: Add_Values, Average, First_Row_PE, Frobenius_Norm, Get_Columns, &
          Get_Values, Infinity_Norm, Last_Row_PE, Max_Nonzeros, MatVec, &
          Maximum, Mean, Minimum, Name, NColumns, NRows_PE, NRows_Total, &
          Norm, One_Norm, Output, Read, Read_Harwell_Boeing, Residual, &
          Set_0_Diagonal_to_1, Set_Not_Up_to_Date, Set_Value, Set_Values, &
          Sum, Total, Two_Norm_Estimate, Two_Norm_Range

interface Initialize
  module procedure Initialize_ELL_Matrix
end interface

interface Finalize
  module procedure Finalize_ELL_Matrix
end interface

interface Valid_State
  module procedure Valid_State_ELL_Matrix
end interface

interface Initialized
  module procedure Initialized_ELL_Matrix
end interface

interface Assignment(=)
  module procedure Get_Columns_ELL_Matrix
  module procedure Get_Values_ELL_Matrix
  module procedure Set_Value_ELL_Matrix_All
end interface

interface Add_Values
  module procedure Add_Values_ELL_Matrix_0
  module procedure Add_Values_ELL_Matrix_1
  module procedure Add_Values_ELL_Matrix_2
end interface

fortext([Value],[Average First_Row_PE Frobenius_Norm Infinity_Norm
                 Last_Row_PE Max_Nonzeros Maximum Minimum Name NColumns
                 NRows_PE NRows_Total One_Norm Sum Two_Norm_Estimate
                 Two_Norm_Range],[
  interface Value
    module procedure expand(Get_Value_ELLM)
  end interface
])

interface Get_Columns
  module procedure Get_Columns_ELL_Matrix
end interface

interface Get_Values
  module procedure Get_Values_ELL_Matrix
end interface
```

```
  interface MatVec
    module procedure MatVec_ELL_Matrix
  end interface

  interface Mean
    module procedure Get_Average_ELLM
  end interface

  interface Norm
    module procedure Get_Frobenius_Norm_ELLM
  end interface

  interface Output
    module procedure Output_ELL_Matrix
  end interface

  interface Read
    module procedure Read_Harwell_Boeing_ELL_Matrix
  end interface

  interface Read_Harwell_Boeing
    module procedure Read_Harwell_Boeing_ELL_Matrix
  end interface

  interface Residual
    module procedure Residual_ELL_Matrix
  end interface

  interface Set_0_Diagonal_to_1
    module procedure Set_0_Diagonal_to_1_ELL_Matrix
  end interface

  interface Set_Not_Up_to_Date
    module procedure Set_Not_Up_to_Date_ELLM
  end interface

  interface Set_Value
    module procedure Set_Value_ELL_Matrix_All
  end interface

  interface Set_Values
    module procedure Set_Values_ELL_Matrix_0
    module procedure Set_Values_ELL_Matrix_1
    module procedure Set_Values_ELL_Matrix_2
    module procedure Set_Values_ELL_Matrix_All
  end interface

  interface Total
    module procedure Get_Sum_ELLM
  end interface

  ! Public type definitions.

  public :: ELL_Matrix_type
```

```
  type ELL_Matrix_type

    ! Initialization status.

    type(integer) :: Initialized

    ! The name for this variable.

    type(character,name_length) :: Name

    ! Basic data structures for Rows and Columns.

    type(Base_Structure_type), pointer :: Row_Structure
    type(Base_Structure_type), pointer :: Column_Structure

    ! Values for the ELL Matrix.

    type(real,2) :: Values

    ! Column numbers for the ELL Matrix.

    type(integer,2) :: Columns

    ! Maximum number of columns (nonzeros) in the matrix.

    type(integer) :: Max_Nonzeros

    ! ELL Matrix dimensionality is two.

    type(integer) :: Dimensionality=2

    ! Data Index and match number that are used for MatVecs.

    type(Data_Index_type) :: Index
    type(integer) :: Index_Match_Number

    ! Norm variables and "updated?" toggles.

    type(real) :: Average, Frobenius_Norm, Infinity_Norm, Maximum, Minimum, &
                  One_Norm, Sum, Two_Norm_Estimate
    type(real), dimension(2) :: Two_Norm_Range
    type(logical) :: Average_is_Updated, Frobenius_Norm_is_Updated, &
                     Index_is_Updated, Infinity_Norm_is_Updated, &
                     Maximum_is_Updated, Minimum_is_Updated, &
                     One_Norm_is_Updated, Sum_is_Updated, &
                     Two_Norm_is_Updated

  end type ELL_Matrix_type

contains
```

The ELL_Matrix Class contains the following routines which are listed in separate sections:

end module Caesar_ELL_Matrix_Class

G.2.1 Initialize_ELL_Matrix Procedure

The main documentation of the Initialize_ELL_Matrix Procedure in § 12.2.1 on page 144 contains additional explanation of this code listing.

```
  subroutine Initialize_ELL_Matrix (ELLM, Max_Nonzeros, Row_Structure, &
                                    Column_Structure, Name, status)

    ! Use association information.

    use Caesar_Flags_Module, only: initialized_flag
    use Caesar_Numbers_Module, only: zero

    ! Input variables.

    type(integer) :: Max_Nonzeros   ! Maximum number of nonzero elements/row.
    ! Row and Column Base_Structures.
    type(Base_Structure_type), intent(in), target :: Row_Structure, &
                                                     Column_Structure
    type(character,*), intent(in), optional :: Name     ! Variable name.

    ! Output variables.

    ! ELL_Matrix to be initialized.
    type(ELL_Matrix_type), intent(out) :: ELLM
```

```
type(Status_type), intent(out), optional :: status  ! Exit status.

! Internal variables.

type(Status_type), dimension(20) :: allocate_status ! Allocation Status.
type(Status_type) :: consolidated_status            ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(.not.Valid_State(ELLM),5)         ! ELL_Matrix is not valid.
VERIFY(Valid_State(Row_Structure),5)     ! Row Structure is valid.
VERIFY(Valid_State(Column_Structure),5)  ! Column Structure is valid.
! Max_Nonzeros is reasonable.
VERIFY(Max_Nonzeros .InInterval. (/1,Length_Total(Column_Structure)/),5)

! Set up structure pointers.

ELLM%Row_Structure => Row_Structure
ELLM%Column_Structure => Column_Structure

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)
call Initialize (ELLM%Values, Length_PE(Row_Structure), Max_Nonzeros, &
                 allocate_status(1))
call Initialize (ELLM%Columns, Length_PE(Row_Structure), Max_Nonzeros, &
                 allocate_status(2))
call Initialize (ELLM%Index, ELLM%Column_Structure, &
                 ELLM%Row_Structure, Many_of_One_Array=ELLM%Columns, &
                 status=allocate_status(3))

! Set up internals.

if (PRESENT(Name)) ELLM%Name = Name
ELLM%Max_Nonzeros = Max_Nonzeros
call Initialize (ELLM%Average,                   allocate_status( 4))
call Initialize (ELLM%Average_is_Updated,        allocate_status( 5))
call Initialize (ELLM%Frobenius_Norm,            allocate_status( 6))
call Initialize (ELLM%Frobenius_Norm_is_Updated, allocate_status( 7))
call Initialize (ELLM%Index_is_Updated,          allocate_status( 8))
call Initialize (ELLM%Infinity_Norm,             allocate_status( 9))
call Initialize (ELLM%Infinity_Norm_is_Updated,  allocate_status(10))
call Initialize (ELLM%Maximum,                   allocate_status(11))
call Initialize (ELLM%Maximum_is_Updated,        allocate_status(12))
call Initialize (ELLM%Minimum,                   allocate_status(13))
call Initialize (ELLM%Minimum_is_Updated,        allocate_status(14))
call Initialize (ELLM%One_Norm,                  allocate_status(15))
call Initialize (ELLM%One_Norm_is_Updated,       allocate_status(16))
call Initialize (ELLM%Sum,                       allocate_status(17))
call Initialize (ELLM%Sum_is_Updated,            allocate_status(18))
call Initialize (ELLM%Two_Norm_Estimate,         allocate_status(19))
```

```
    call Initialize (ELLM%Two_Norm_is_Updated,       allocate_status(20))

    ! Make sure that initial values are correct.

    ELLM%Average = zero
    ELLM%Average_is_Updated = .false.
    ELLM%Frobenius_Norm = zero
    ELLM%Frobenius_Norm_is_Updated = .false.
    ELLM%Index_is_Updated = .false.
    ELLM%Infinity_Norm = zero
    ELLM%Infinity_Norm_is_Updated = .false.
    ELLM%Maximum = zero
    ELLM%Maximum_is_Updated = .false.
    ELLM%Minimum = zero
    ELLM%Minimum_is_Updated = .false.
    ELLM%One_Norm = zero
    ELLM%One_Norm_is_Updated = .false.
    ELLM%Sum = zero
    ELLM%Sum_is_Updated = .false.
    ELLM%Two_Norm_Estimate = zero
    ELLM%Two_Norm_Range = (/ zero, zero /)
    ELLM%Two_Norm_is_Updated = .false.

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set initialization flag.

    ELLM%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)     ! ELL_Matrix is now valid.

    return
  end subroutine Initialize_ELL_Matrix
```

G.2.2 Finalize_ELL_Matrix Procedure

The main documentation of the Finalize_ELL_Matrix Procedure in § 12.2.2 on page 144 contains additional explanation of this code listing.

```
  subroutine Finalize_ELL_Matrix (ELLM, status)
```

```
! Use association information.

use Caesar_Flags_Module, only: uninitialized_flag
use Caesar_Numbers_Module, only: zero

! Input/Output variable.

! ELL_Matrix to be finalized.
type(ELL_Matrix_type), intent(inout) :: ELLM

! Output variables.

type(Status_type), intent(out), optional :: status   ! Exit status.

! Internal variables.

! Deallocation Status.
type(Status_type), dimension(22) :: deallocate_status
type(Status_type) :: consolidated_status             ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(ELLM),7)                   ! ELL_Matrix is valid.

! Unset initialization flag.

ELLM%Initialized = uninitialized_flag

! Deallocations and finalizations.

! Set deallocation status.

call Initialize (deallocate_status)
call Initialize (consolidated_status)

! Finalize internals.

NULLIFY(ELLM%Row_Structure)
NULLIFY(ELLM%Column_Structure)
call Finalize (ELLM%Average,                     deallocate_status( 1))
call Finalize (ELLM%Average_is_Updated,          deallocate_status( 2))
call Finalize (ELLM%Columns,                     deallocate_status( 3))
call Finalize (ELLM%Frobenius_Norm,              deallocate_status( 4))
call Finalize (ELLM%Frobenius_Norm_is_Updated,   deallocate_status( 5))
call Finalize (ELLM%Index,                       deallocate_status( 6))
call Finalize (ELLM%Index_is_Updated,            deallocate_status( 7))
call Finalize (ELLM%Infinity_Norm,               deallocate_status( 8))
call Finalize (ELLM%Infinity_Norm_is_Updated,    deallocate_status( 9))
call Finalize (ELLM%Max_Nonzeros,                deallocate_status(10))
call Finalize (ELLM%Maximum,                     deallocate_status(11))
call Finalize (ELLM%Maximum_is_Updated,          deallocate_status(12))
```

```
    call Finalize (ELLM%Minimum,                   deallocate_status(13))
    call Finalize (ELLM%Minimum_is_Updated,        deallocate_status(14))
    call Finalize (ELLM%Name,                      deallocate_status(15))
    call Finalize (ELLM%One_Norm,                  deallocate_status(16))
    call Finalize (ELLM%One_Norm_is_Updated,       deallocate_status(17))
    call Finalize (ELLM%Sum,                       deallocate_status(18))
    call Finalize (ELLM%Sum_is_Updated,            deallocate_status(19))
    call Finalize (ELLM%Two_Norm_Estimate,         deallocate_status(20))
    call Finalize (ELLM%Two_Norm_is_Updated,       deallocate_status(21))
    call Finalize (ELLM%Values,                    deallocate_status(22))
    ELLM%Two_Norm_Range = (/ zero, zero /)

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(ELLM),7)          ! ELL_Matrix is not valid.

    return
  end subroutine Finalize_ELL_Matrix
```

G.2.3 Valid_State_ELL_Matrix Procedure

The main documentation of the Valid_State_ELL_Matrix Procedure in § 12.2.3 on page 145 contains additional explanation of this code listing.

```
  function Valid_State_ELL_Matrix (ELLM) result(Valid)

    ! Use association information.

    use Caesar_Numbers_Module, only: zero

    ! Input variables.

    ! Variable to be checked.
    type(ELL_Matrix_type), intent(in) :: ELLM

    ! Output variables.

    type(logical) :: Valid              ! Logical state.

    ! Internal variables.
```

```
type(real) :: N, M                 ! Dimensions of the ELLM.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Start out true.

Valid = .true.

! Check for association of pointered internals.

Valid = Valid .and. ASSOCIATED(ELLM%Row_Structure)
Valid = Valid .and. ASSOCIATED(ELLM%Column_Structure)
if (.not.Valid) return

! Check for validity of internals.

Valid = Valid .and. Initialized(ELLM)
Valid = Valid .and. Valid_State(ELLM%Average)
Valid = Valid .and. Valid_State(ELLM%Average_is_Updated)
Valid = Valid .and. Valid_State(ELLM%Column_Structure)
Valid = Valid .and. Valid_State(ELLM%Frobenius_Norm)
Valid = Valid .and. Valid_State(ELLM%Frobenius_Norm_is_Updated)
Valid = Valid .and. Valid_State(ELLM%Infinity_Norm)
Valid = Valid .and. Valid_State(ELLM%Infinity_Norm_is_Updated)
Valid = Valid .and. Valid_State(ELLM%Max_Nonzeros)
Valid = Valid .and. Valid_State(ELLM%Maximum)
Valid = Valid .and. Valid_State(ELLM%Maximum_is_Updated)
Valid = Valid .and. Valid_State(ELLM%Minimum)
Valid = Valid .and. Valid_State(ELLM%Minimum_is_Updated)
Valid = Valid .and. Valid_State(ELLM%Name)
Valid = Valid .and. Valid_State(ELLM%One_Norm)
Valid = Valid .and. Valid_State(ELLM%One_Norm_is_Updated)
Valid = Valid .and. Valid_State(ELLM%Row_Structure)
Valid = Valid .and. Valid_State(ELLM%Sum)
Valid = Valid .and. Valid_State(ELLM%Sum_is_Updated)
Valid = Valid .and. Valid_State(ELLM%Two_Norm_Estimate)
Valid = Valid .and. Valid_State(ELLM%Two_Norm_is_Updated)
if (.not.Valid) return

! Checks on the validity of ELL_Matrix.

N = changetype(real, Length_Total(ELLM%Row_Structure))
M = changetype(real, Length_Total(ELLM%Column_Structure))
Valid = Valid .and. ALL(ELLM%Columns >= 0)
Valid = Valid .and. ELLM%Infinity_Norm >= zero
Valid = Valid .and. ELLM%One_Norm >= zero
Valid = Valid .and. ELLM%Frobenius_Norm >= zero
Valid = Valid .and. ELLM%Two_Norm_Estimate >= zero
if (ELLM%Two_Norm_is_Updated) then
  Valid = Valid .and. &
          (ELLM%Two_Norm_Estimate .InInterval. ELLM%Two_Norm_Range)
end if
if (ELLM%Average_is_updated .and. ELLM%Sum_is_updated) then
```

```
      Valid = Valid .and. &
              (ELLM%Average .VeryClose. ELLM%Sum/changetype(real,(N*M)))
    end if
    if (ELLM%Maximum_is_updated .and. ELLM%Minimum_is_updated) then
      Valid = Valid .and. ELLM%Maximum >= ELLM%Minimum
    end if
    if (.not.Valid) return

    ! Mathematic relationship checks.

    !if (ELLM%One_Norm_is_updated .and. ELLM%Two_Norm_is_updated) then
    !  Valid = Valid .and. ELLM%One_Norm >= ELLM%Two_Norm
    !  Valid = Valid .and. ELLM%Two_Norm * SQRT(N) >= ELLM%One_Norm
    !end if
    !if (ELLM%Infinity_Norm_is_updated .and. ELLM%Two_Norm_is_updated) then
    !  Valid = Valid .and. ELLM%Two_Norm >= ELLM%Infinity_Norm
    !  Valid = Valid .and. ELLM%Infinity_Norm * SQRT(N) >= ELLM%Two_Norm
    !end if
    !if (ELLM%Infinity_Norm_is_updated .and. ELLM%One_Norm_is_updated) then
    !  Valid = Valid .and. ELLM%One_Norm >= ELLM%Infinity_Norm
    !  Valid = Valid .and. ELLM%Infinity_Norm * N >= ELLM%One_Norm
    !end if

    return
  end function Valid_State_ELL_Matrix
```

G.2.4 Initialized_ELL_Matrix Procedure

The main documentation of the Initialized_ELL_Matrix Procedure in § 12.2.4 on page 145 contains additional explanation of this code listing.

```
  function Initialized_ELL_Matrix (ELLM) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! ELL_Matrix to be checked.
    type(ELL_Matrix_type), intent(in) :: ELLM

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.
```

```
    Initialized = ELLM%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_ELL_Matrix
```

G.2.5 Add_Values_ELL_Matrix Procedure

The main documentation of the Add_Values_ELL_Matrix Procedure in § 12.2.5 on page 146 contains additional explanation of this code listing.

```
  subroutine Add_Values_ELL_Matrix_0 (ELLM, Value, Row, Column, Global)

    ! Note: this routine is very similar to Set_Values_ELL_Matrix_0.

    ! Input variable.

    type(real), intent(in) :: Value               ! Value scalar.
    type(integer), intent(in) :: Row              ! Row integer scalar.
    type(integer), intent(in) :: Column           ! Column integer scalar.
    type(logical), intent(in), optional :: Global ! Global/local index toggle.

    ! Input/Output variable.

    type(ELL_Matrix_type), intent(inout) :: ELLM  ! Variable to be incremented.

    ! Internal variables.

    type(logical) :: A_Global        ! Actual global/local toggle.
    type(integer) :: Column_Location ! Location in Columns array for the entry.
    type(integer) :: location        ! Loop index.
    type(integer) :: shift           ! Index shift.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)                         ! ELLM is valid.
    VERIFY(Valid_State(Value),5)                        ! Value is valid.
    VERIFY(Valid_State(Row),5)                          ! Row is valid.
    VERIFY(Valid_State(Column),5)                       ! Column is valid.

    ! Global/Local toggle.

    if (PRESENT(Global)) then
      A_Global = Global
    else
      A_Global = .true.
    end if
    if (A_Global) then
      shift = -First_PE(ELLM%Row_Structure) + 1
```

```
    else
      shift = 0
    end if

    ! Another requirement check -- require that Row be in the correct range.

    VERIFY(Row + shift .InInterval. (/1, Length_PE(ELLM%Row_Structure)/),5)

    ! Add the value.

    if (Row /= 0) then

      ! Find a column location to store the entry.

      Column_Location = 0
      do location = 1, ELLM%Max_Nonzeros
        if (ELLM%Columns(Row + shift, location) == 0 .or. &
            ELLM%Columns(Row + shift, location) == Column) then
          Column_Location = location
          exit
        end if
      end do

      ! Add the entry.

      ELLM%Values(Row + shift, Column_Location) = &
        ELLM%Values(Row + shift, Column_Location) + Value
      ELLM%Columns(Row + shift, Column_Location) = Column

    else

      ! Make sure Column_Location has a non-zero value if Row=0,
      ! so the check below executes correctly.
      Column_Location = -1

    end if

    ! Make sure Max_Nonzeros is not exceeded (that is, that we
    ! found a spot to put the entry).

    VERIFY(Column_Location /= 0,1)

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (ELLM)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                    ! ELLM is still valid.

    return
  end subroutine Add_Values_ELL_Matrix_0

  subroutine Add_Values_ELL_Matrix_1 (ELLM, Values, Rows, Columns, Global)
```

```
! Note: this routine is very similar to Set_Values_ELL_Matrix_1.

! Input variable.

type(real,1,np), intent(in) :: Values          ! Values bare naked array.
type(integer,1,np), intent(in) :: Rows         ! Rows integer vector.
type(integer,1,np), intent(in) :: Columns      ! Columns integer vector.
type(logical), intent(in), optional :: Global  ! Global/local index toggle.

! Input/Output variable.

type(ELL_Matrix_type), intent(inout) :: ELLM ! Variable to be incremented.

! Internal variables.

type(logical) :: A_Global                   ! Actual global/local toggle.
type(integer) :: Column_Location            ! Location in Columns array
                                            !   for the entry.
type(integer) :: i                          ! Loop parameter.
type(integer) :: location                   ! Loop index.
type(logical) :: Max_Nonzeros_Not_Exceeded  ! Toggle for error check.
type(integer) :: shift                      ! Index shift.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(ELLM),5)                         ! ELLM is valid.
VERIFY(Valid_State_NP(Values),5)                    ! Values is valid.
VERIFY(Valid_State_NP(Rows),5)                      ! Rows is valid.
VERIFY(Valid_State_NP(Columns),5)                   ! Columns is valid.
! Values, Rows & Columns size checks.
VERIFY(SIZE(Values,1) <= Length_PE(ELLM%Row_Structure),5)
VERIFY(SIZE(Rows) <= Length_PE(ELLM%Row_Structure),5)
VERIFY(SIZE(Rows) == SIZE(Values,1),5)
VERIFY(SIZE(Values) == SIZE(Columns),5)

! Global/Local toggle.

if (PRESENT(Global)) then
  A_Global = Global
else
  A_Global = .true.
end if
if (A_Global) then
  shift = -First_PE(ELLM%Row_Structure) + 1
else
  shift = 0
end if

! More requirement checks -- require that Rows entries are in the
! correct range.
```

```
    VERIFY(Rows + shift .InInterval. (/1, Length_PE(ELLM%Row_Structure)/),5)

    ! Add the values.

    Max_Nonzeros_Not_Exceeded = .true.
    do i = 1, SIZE(Rows)
      if (Rows(i) /= 0) then

        ! Find a column location to store the entry.

        Column_Location = 0
        do location = 1, ELLM%Max_Nonzeros
          if (ELLM%Columns(Rows(i) + shift, location) == 0 .or. &
              ELLM%Columns(Rows(i) + shift, location) == Columns(i)) then
            Column_Location = location
            exit
          end if
        end do

        ! Make sure Max_Nonzeros is not exceeded (that is, that we
        ! found a spot to put the entry).

        Max_Nonzeros_Not_Exceeded = &
          Max_Nonzeros_Not_Exceeded .and. Column_Location /= 0

        ! Increment the entry.

        ELLM%Values(Rows(i) + shift, Column_Location) = &
          ELLM%Values(Rows(i) + shift, Column_Location) + Values(i)
        ELLM%Columns(Rows(i) + shift, Column_Location) = Columns(i)

      end if
    end do

    ! Make sure Max_Nonzeros is not exceeded (that is, that we
    ! found a spot to put all of the entries).

    VERIFY(Max_Nonzeros_Not_Exceeded,1)

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (ELLM)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                   ! ELLM is still valid.

    return
  end subroutine Add_Values_ELL_Matrix_1

  subroutine Add_Values_ELL_Matrix_2 (ELLM, Values, Rows, Columns, Global)

    ! Note: this routine is very similar to Set_Values_ELL_Matrix_2.
```

```
! Input variable.

type(real,2,np), intent(in) :: Values          ! Values bare naked array.
type(integer,1,np), intent(in) :: Rows         ! Rows integer vector.
type(integer,2,np), intent(in) :: Columns      ! Columns integer array.
type(logical), intent(in), optional :: Global  ! Global/local index toggle.

! Input/Output variable.

type(ELL_Matrix_type), intent(inout) :: ELLM ! Variable to be incremented.

! Internal variables.

type(logical) :: A_Global                   ! Actual global/local toggle.
type(integer) :: Column_Location            ! Location in Columns array
                                            !   for the entry.
type(integer) :: i, j                       ! Loop parameters.
type(integer) :: location                   ! Loop index.
type(logical) :: Max_Nonzeros_Not_Exceeded  ! Toggle for error check.
type(integer) :: shift                      ! Index shift.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(ELLM),5)                         ! ELLM is valid.
VERIFY(Valid_State_NP(Values),5)                    ! Values is valid.
VERIFY(Valid_State_NP(Rows),5)                      ! Rows is valid.
VERIFY(Valid_State_NP(Columns),5)                   ! Columns is valid.
! Values, Rows & Columns size checks.
VERIFY(SIZE(Values,1) <= Length_PE(ELLM%Row_Structure),5)
VERIFY(SIZE(Values,2) <= ELLM%Max_Nonzeros,5)
VERIFY(SIZE(Rows) <= Length_PE(ELLM%Row_Structure),5)
VERIFY(SIZE(Rows) == SIZE(Values,1),5)
VERIFY(SIZE(Values) == SIZE(Columns),5)

! Global/Local toggle.

if (PRESENT(Global)) then
  A_Global = Global
else
  A_Global = .true.
end if
if (A_Global) then
  shift = -First_PE(ELLM%Row_Structure) + 1
else
  shift = 0
end if

! More requirement checks -- require that Rows entries are in the
! correct range.

VERIFY(Rows + shift .InInterval. (/1, Length_PE(ELLM%Row_Structure)/),5)
```

```
    ! Add the values.

    Max_Nonzeros_Not_Exceeded = .true.
    do i = 1, SIZE(Rows)
      if (Rows(i) /= 0) then
        do j = 1, SIZE(Values,2)

          ! Find a column location to store the entry.

          Column_Location = 0
          do location = 1, ELLM%Max_Nonzeros
            if (ELLM%Columns(Rows(i) + shift, location) == 0 .or. &
                ELLM%Columns(Rows(i) + shift, location) == Columns(i,j)) then
              Column_Location = location
              exit
            end if
          end do

          ! Make sure Max_Nonzeros is not exceeded (that is, that we
          ! found a spot to put the entry).

          Max_Nonzeros_Not_Exceeded = &
            Max_Nonzeros_Not_Exceeded .and. Column_Location /= 0

          ! Increment the entry.

          ELLM%Values(Rows(i) + shift, Column_Location) = &
            ELLM%Values(Rows(i) + shift, Column_Location) + Values(i,j)
          ELLM%Columns(Rows(i) + shift, Column_Location) = Columns(i,j)

        end do
      end if
    end do

    ! Make sure Max_Nonzeros is not exceeded (that is, that we
    ! found a spot to put all of the entries).

    VERIFY(Max_Nonzeros_Not_Exceeded,1)

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (ELLM)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                    ! ELLM is still valid.

    return
  end subroutine Add_Values_ELL_Matrix_2
```

G.2.6 Get Value ELL_Matrix Functions

The main documentation of the Get Value ELL_Matrix Functions in § 12.2.6 on page 146 contains additional explanation of this code listing.

```
define([REAL_ACCESS_ROUTINE],[
  pushdef([VALUE], [$1])
  pushdef([Get_REAL_VALUE_ELLM], expand(Get_VALUE_ELLM))
  ifelse(
    VALUE, [Two_Norm_Estimate],
      [pushdef([VALUE_is_Updated], Two_Norm_is_Updated)],
    VALUE, [Two_Norm_Range],
      [pushdef([VALUE_is_Updated], Two_Norm_is_Updated)],
    [pushdef([VALUE_is_Updated], expand(VALUE_is_Updated))])

  function Get_REAL_VALUE_ELLM (ELLM) result(VALUE)

    ! Use association information.

    ifelse(VALUE, [Two_Norm_Estimate], [
      use Caesar_Numbers_Module, only: half
    ])

    ! Input/Output variables.

    ! ELL_Matrix object.
    type(ELL_Matrix_type), intent(inout) :: ELLM

    ! Output variables.

    ifelse(
      VALUE, [Two_Norm_Range],
        [type(real), dimension(2) :: VALUE ! ELL_Matrix value to be output.],
      [type(real) :: VALUE                 ! ELL_Matrix value to be output.])

    ! Internal variables.

    ifelse(
      VALUE, [One_Norm],
        [type(real,1) :: Column_Sums          ! Column sum temporary.    ],
      VALUE, [Two_Norm_Estimate],
        [type(real), dimension(2) :: TNRange  ! Two Norm temporary.      ],
      VALUE, [Two_Norm_Range],
        [type(real) :: N                      ! Matrix row dimension.
         type(real) :: M                      ! Matrix column dimension. ],
      [])

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)       ! ELL_Matrix is valid.
```

```
    ! Set value.

    if (ELLM%VALUE_is_Updated) then
      VALUE = ELLM%VALUE
    else
      ifelse(
        VALUE, [Average],
          [VALUE = Global_Sum(ELLM%Values) / &
                   Length_Total(ELLM%Row_Structure) / &
                   Length_Total(ELLM%Column_Structure) ],
        VALUE, [Frobenius_Norm],
          [VALUE = SQRT(Global_Sum(ELLM%Values**2))],
        VALUE, [Infinity_Norm],
          [VALUE = Global_MaxVal(SUM(ABS(ELLM%Values),2))],
        VALUE, [Maximum],
          [VALUE = Global_MaxVal(ELLM%Values)],
        VALUE, [Minimum],
          [VALUE = Global_MinVal(ELLM%Values)],
        VALUE, [One_Norm],
          [call Initialize (Column_Sums, Length_PE(ELLM%Column_Structure))
           call Scatter_SUM (Column_Sums, ABS(ELLM%Values), ELLM%Columns)
           VALUE = Global_MaxVal(Column_Sums)
           call Finalize (Column_Sums)],
        VALUE, [Two_Norm_Estimate],
          [TNRange = Two_Norm_Range(ELLM)
           VALUE = half * (TNRange(1) + TNRange(2))],
        VALUE, [Two_Norm_Range],
          [N = changetype(real, Length_Total(ELLM%Row_Structure))
           M = changetype(real, Length_Total(ELLM%Column_Structure))
           VALUE[](1) = MAX(Frobenius_Norm(ELLM)/SQRT(N), &
                            Maximum(ELLM),               &
                            Infinity_Norm(ELLM)/SQRT(N), &
                            One_Norm(ELLM)/SQRT(M))
           VALUE[](2) = MIN(Frobenius_Norm(ELLM),        &
                            SQRT(M*N)*Maximum(ELLM),     &
                            SQRT(M)*Infinity_Norm(ELLM), &
                            SQRT(N)*One_Norm(ELLM),      &
                            SQRT(One_Norm(ELLM)*Infinity_Norm(ELLM)))],
        VALUE, [Sum],
          [VALUE = Global_Sum(ELLM%Values)],
        [])
      ELLM%VALUE_is_Updated = .true.
      ELLM%VALUE = VALUE
    end if

    ! Verify guarantees - none.

    return
  end function Get_REAL_VALUE_ELLM

  popdef([VALUE])
  popdef([Get_REAL_VALUE_ELLM])
  popdef([VALUE_is_Updated])
])
```

```
fortext([Value],
        [Average Frobenius_Norm Infinity_Norm Maximum Minimum One_Norm Sum
         Two_Norm_Estimate Two_Norm_Range],[
  REAL_ACCESS_ROUTINE(Value)
])

define([INTEGER_ACCESS_ROUTINE],[
  pushdef([VALUE], [$1])
  pushdef([VALUE_Result], expand(VALUE_Result))
  pushdef([Get_INTEGER_VALUE_ELLM], expand(Get_VALUE_ELLM))

  function Get_INTEGER_VALUE_ELLM (ELLM) result(VALUE_Result)

    ! Input/Output variables.

    ! ELL_Matrix object.
    type(ELL_Matrix_type), intent(in) :: ELLM

    ! Output variables.

    type(integer) :: VALUE_Result     ! ELL_Matrix value to be output.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)       ! ELL_Matrix is valid.

    ! Set value.

    ifelse(
      VALUE, [First_Row_PE],
        [VALUE_Result = First_PE(ELLM%Row_Structure)],
      VALUE, [Last_Row_PE],
        [VALUE_Result = Last_PE(ELLM%Row_Structure)],
      VALUE, [Max_Nonzeros],
        [VALUE_Result = ELLM%Max_Nonzeros],
      VALUE, [NColumns],
        [VALUE_Result = Length_Total(ELLM%Column_Structure)],
      VALUE, [NRows_PE],
        [VALUE_Result = Length_PE(ELLM%Row_Structure)],
      VALUE, [NRows_Total],
        [VALUE_Result = Length_Total(ELLM%Row_Structure)],
      [])

    ! Verify guarantees - none.

    return
  end function Get_INTEGER_VALUE_ELLM

  popdef([VALUE])
  popdef([VALUE_Result])
  popdef([Get_INTEGER_VALUE_ELLM])
```

```
  ])

  fortext([Value],
          [First_Row_PE Last_Row_PE Max_Nonzeros NColumns NRows_PE
           NRows_Total],[
    INTEGER_ACCESS_ROUTINE(Value)
  ])

  function Get_Name_ELLM (ELLM) result(Name)

    ! Input variables.

    ! ELL_Matrix object.
    type(ELL_Matrix_type), intent(in) :: ELLM

    ! Output variables.

    type(character,80) :: Name      ! ELL_Matrix value to be output.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)       ! ELL_Matrix is valid.

    ! Set value.

    Name = ELLM%Name

    ! Verify guarantees - none.

    return
  end function Get_Name_ELLM
```

G.2.7 Get_Columns_ELL_Matrix Procedure

The main documentation of the Get_Columns_ELL_Matrix Procedure in § 12.2.7 on page 147 contains additional explanation of this code listing.

```
  subroutine Get_Columns_ELL_Matrix (Columns, ELLM)

    ! Input variable.

    type(ELL_Matrix_type), intent(in) :: ELLM     ! Matrix to be queried.

    ! Input/Output variable.

    type(integer,2,np), intent(inout) :: Columns  ! Columns bare naked array.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.
```

```
    VERIFY(Valid_State(ELLM),5)                          ! ELLM is valid.
    VERIFY(Valid_State_NP(Columns),5)                    ! Columns is valid.
    ! Columns size checks.
    VERIFY(SIZE(Columns,1) == Length_PE(ELLM%Row_Structure),5)
    VERIFY(SIZE(Columns,2) == ELLM%Max_Nonzeros,5)

    ! Get the columns.

    Columns = ELLM%Columns

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                ! ELLM is still valid.

    return
  end subroutine Get_Columns_ELL_Matrix
```

G.2.8 Get_Values_ELL_Matrix Procedure

The main documentation of the Get_Values_ELL_Matrix Procedure in § 12.2.8 on page 148 contains additional explanation of this code listing.

```
  subroutine Get_Values_ELL_Matrix (Values, ELLM)

    ! Input variable.

    type(ELL_Matrix_type), intent(in) :: ELLM     ! Matrix to be queried.

    ! Input/Output variable.

    type(real,2,np), intent(inout) :: Values      ! Values bare naked array.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)                         ! ELLM is valid.
    VERIFY(Valid_State_NP(Values),5)                    ! Values is valid.
    ! Values size checks.
    VERIFY(SIZE(Values,1) == Length_PE(ELLM%Row_Structure),5)
    VERIFY(SIZE(Values,2) == ELLM%Max_Nonzeros,5)

    ! Get the values.

    Values = ELLM%Values

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                           ! ELLM is still valid.

    return
```

```
  end subroutine Get_Values_ELL_Matrix
```

G.2.9 MatVec_ELL_Matrix Procedure

The main documentation of the MatVec_ELL_Matrix Procedure in § 12.2.9 on page 148 contains additional explanation of this code listing.

```
  subroutine MatVec_ELL_Matrix (ELLM, MV_in, MV_out)

    ! Input variables.

    ! ELL Matrix to be multiplied.
    type(ELL_Matrix_type), intent(inout) :: ELLM
    ! Mathematic Vector to be multiplied.
    type(Mathematic_Vector_type), intent(inout) :: MV_in

    ! Output variable.

    ! Result of dot product.
    type(Mathematic_Vector_type), intent(inout) :: MV_out

    ! Internal variables.

    type(integer) :: i                     ! Loop counter.
    type(integer) :: OV_match              ! The OV match for ELLM.
    type(real,2) :: MV_in_Values_Array     ! Temp for collected MV_in Values.
    type(real) :: random_real              ! Random temporary.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)            ! ELLM is valid.
    VERIFY(Valid_State(MV_in),5)           ! MV_in is valid.
    VERIFY(Valid_State(MV_out),5)          ! MV_out is valid.
    ! MV_in has same Structure as ELLM Column_Structure.
    VERIFY(ASSOCIATED(ELLM%Column_Structure,MV_in%Structure),5)
    ! MV_out has same Structure as ELLM Row_Structure.
    VERIFY(ASSOCIATED(ELLM%Row_Structure,MV_out%Structure),5)
    VERIFY(Number_of_OVs_in_an_MV==4,5)    ! 4 is assumed in this procedure.

    ! Check to see if Index needs to be updated.

    if (.not. ELLM%Index_is_Updated) then
      call Finalize (ELLM%Index)
      call Initialize (ELLM%Index, ELLM%Column_Structure, &
                       ELLM%Row_Structure, Many_of_One_Array=ELLM%Columns)
      ELLM%Index_is_Updated = .true.
      call RANDOM_NUMBER (random_real)
      ELLM%Index_Match_Number = changetype(integer, random_real*1.e9)
    end if
```

```
! ### SPRONG -- need to step through the logic in this routine,
! ### especially to make sure that MV_in%DV is updated correctly
! ### (and not updated when it shouldn't be).

! Check to see if MV_in already has an Overlapped_Vector for this Matrix.

OV_match = 0
do i = 1, Number_of_OVs_in_an_MV
  if (Initialized(MV_in%OV(i))) then
   if (ELLM%Index_Match_Number == MV_in%Index_Match_Number(i)) then
      OV_match = i
    end if
  end if
end do

! Create MV_in%OV, MV_in%DV if needed.

if (OV_match == 0) then

  ! Determine which slot to put new OV in.

  do i = 1, Number_of_OVs_in_an_MV
    if (Initialized(MV_in%OV(i))) then
      OV_match = i + 1
    end if
  end do
  if (OV_match == 0 .or. OV_match == 5) then
    OV_match = 1
  end if

  ! Set up MV_in%DV if necessary.

  if (.not.Initialized(MV_in%DV)) then
    call Initialize (MV_in%DV, MV_in%Structure, MV_in%Dimensionality, &
                     MV_in%Name)
  end if

  ! Set up MV_in%OV(OV_match).

  if (Initialized(MV_in%OV(OV_match))) call Finalize (MV_in%OV(OV_match))
  MV_in%DV = MV_in%Values
  call Initialize (MV_in%OV(OV_match), MV_in%DV, ELLM%Index, MV_in%Name)
  MV_in%Index_Match_Number(OV_match) = ELLM%Index_Match_Number

end if

! Update MV_in%DV and MV_in%OV if needed.

call Update_DV (MV_in)
MV_in%OV(OV_match) = MV_in%DV

! Extract the values from MV_in%OV.

call Initialize (MV_in_Values_Array, Length_PE(MV_in%Structure), &
```

```
                    ELLM%Max_Nonzeros)
    MV_in_Values_Array = MV_in%OV(OV_match)

    ! Calculate the global matrix-vector product.

    MV_out%Values(:) = SUM(ELLM%Values * MV_in_Values_Array, 2)

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (MV_out)

    ! Clean up.

    call Finalize (MV_in_Values_Array)

    ! Verify guarantees - none.

    return
  end subroutine MatVec_ELL_Matrix
```

G.2.10 Output_ELL_Matrix Procedure

The main documentation of the Output_ELL_Matrix Procedure in § 12.2.10 on page 149 contains additional explanation of this code listing.

```
  subroutine Output_ELL_Matrix (ELLM, Row_First, Row_Last, Unit, Indent)

    ! Input variables.

    ! Variable to be output.
    type(ELL_Matrix_type), intent(inout) :: ELLM
    type(integer), intent(in), optional :: Row_First  ! Extents of value data
    type(integer), intent(in), optional :: Row_Last   !   to be output.
    type(integer), intent(in), optional :: Unit       ! Output unit.
    type(integer), optional :: Indent                 ! Indentation.

    ! Internal variables.

    type(integer) :: Buffer_Loc                       ! Buffer location.
    type(integer) :: Buffer_Size                      ! Output buffer size.
    type(integer) :: Buffer_Skip                      ! Buffer increment.
    type(integer) :: i_global, i_local                ! Loop counters.
    type(integer) :: A_Row_First                      ! Actual first row value.
    type(integer) :: A_Row_Last                       ! Actual last row value.
    type(integer) :: A_Unit                           ! Actual output unit.
    type(integer) :: A_Indent                         ! Actual indentation.
    type(character,80) :: Blanks                      ! A line of blanks.
    type(character,80) :: ELLM_Name                   ! Name of the ELLM.
    type(character,80) :: Output_1                    ! Output buffer.
    type(character,80,1) :: Output_Buffer             ! Output buffer vector.
    type(real) :: ELLM_Average, ELLM_Frobenius_Norm, & ! Get Value variables.
                  ELLM_Infinity_Norm, ELLM_Maximum, &
```

```
              ELLM_Minimum, ELLM_One_Norm, &
              ELLM_Sum, ELLM_Two_Norm_Estimate
type(real), dimension(2) :: ELLM_Two_Norm_Range

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(ELLM),5)      ! ELLM is valid.

! Set unit number.

if (PRESENT(Unit)) then
  A_Unit = Unit
else
  A_Unit = 6
end if

! Set indentation.

if (PRESENT(Indent)) then
  A_Indent = Indent
else
  A_Indent = 0
end if
Blanks = ' '

! These are evaluated on all PEs -- NOT inside an IO PE block --
! because they contain validity checks on ELLM and thus require
! global communication.

ELLM_Name = Name(ELLM)
ELLM_Average = Average(ELLM)
ELLM_Infinity_Norm = Infinity_Norm(ELLM)
ELLM_Maximum = Maximum(ELLM)
ELLM_Minimum = Minimum(ELLM)
ELLM_One_Norm = One_Norm(ELLM)
ELLM_Frobenius_Norm = Frobenius_Norm(ELLM)
ELLM_Sum = Sum(ELLM)
ELLM_Two_Norm_Estimate = Two_Norm_Estimate(ELLM)
ELLM_Two_Norm_Range = Two_Norm_Range(ELLM)

! Output Identification Info.

if (this_is_IO_PE) then
  write (A_Unit,100) Blanks(1:A_Indent), 'ELL Matrix Information:'
  write (A_Unit,101) Blanks(1:A_Indent+2), 'Name                = ', &
                     TRIM(ELLM_Name)
  write (A_Unit,102) Blanks(1:A_Indent+2), 'Initialized         = ', &
                     Initialized(ELLM)
  write (A_Unit,103) Blanks(1:A_Indent+2), 'Dimensionality      =', &
                     ELLM%Dimensionality
  write (A_Unit,103) Blanks(1:A_Indent+2), 'Max_Nonzeros        =', &
                     ELLM%Max_Nonzeros
```

```
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Average              =', &
                         ELLM_Average
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Maximum              =', &
                         ELLM_Maximum
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Minimum              =', &
                         ELLM_Minimum
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Sum                  =', &
                         ELLM_Sum
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Infinity_Norm        =', &
                         ELLM_Infinity_Norm
      write (A_Unit,104) Blanks(1:A_Indent+2), 'One_Norm             =', &
                         ELLM_One_Norm
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Two_Norm_Estimate    =', &
                         ELLM_Two_Norm_Estimate
      write (A_Unit,105) Blanks(1:A_Indent+2), 'Two_Norm_Range       =', &
                         ELLM_Two_Norm_Range
      write (A_Unit,104) Blanks(1:A_Indent+2), 'Frobenius_Norm       =', &
                         ELLM_Frobenius_Norm
    end if

    ! Output internal structure info.

    call Output (ELLM%Row_Structure, A_Unit, 'Row', A_Indent+2)
    call Output (ELLM%Column_Structure, A_Unit, 'Column', A_Indent+2)

    ! Output internal values.

    if (this_is_IO_PE) then
      write (A_Unit,100) Blanks(1:A_Indent), '  Internal Values:'
    end if

    ! Set up local limits in terms of global limits.

    if (PRESENT(Row_First)) then
      A_Row_First = Row_First
    else
      A_Row_First = 1
    end if
    if (PRESENT(Row_Last)) then
      A_Row_Last = Row_Last
    else
      A_Row_Last = Length_Total(ELLM%Row_Structure)
    end if
    A_Row_First = MAX(A_Row_First, First_PE(ELLM%Row_Structure))
    A_Row_Last = MIN(A_Row_Last, Last_PE(ELLM%Row_Structure))

    ! Output Values.

    Buffer_Size = MAX(0, ((ELLM%Max_Nonzeros + 2) / 3) &
                  * (A_Row_Last - A_Row_First + 1))
    call Initialize (Output_Buffer, Buffer_Size)
    if (Buffer_Size /= 0) then
      Buffer_Skip = (ELLM%Max_Nonzeros + 2) / 3
      Buffer_Loc = 1
```

```
      do i_global = A_Row_First, A_Row_Last
        i_local = i_global - First_PE(ELLM%Row_Structure) + 1
        write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),106) &
          'PE:', this_PE, ', Values(', i_global, ',:) =', &
          ELLM%Values(i_local,:)
        Buffer_Loc = Buffer_Loc + Buffer_Skip
      end do
    end if

    ! Add indentation and output.

    do Buffer_loc = 1, Buffer_Size
      Output_1 = Output_Buffer(Buffer_loc)
      Output_Buffer(Buffer_loc) = Blanks(1:A_Indent) // Output_1
    end do
    call Parallel_Write (Output_Buffer, A_Unit)

    ! Output Columns.

    if (this_is_IO_PE) then
      write (A_Unit,*) ' '
    end if
    if (Buffer_Size /= 0) then
      Buffer_Skip = (ELLM%Max_Nonzeros + 2) / 3
      Buffer_Loc = 1
      do i_global = A_Row_First, A_Row_Last
        i_local = i_global - First_PE(ELLM%Row_Structure) + 1
        write (Output_Buffer(Buffer_Loc:Buffer_Loc+Buffer_Skip-1),107) &
          'PE:', this_PE, ', Columns(', i_global, ',:) =', &
          ELLM%Columns(i_local,:)
        Buffer_Loc = Buffer_Loc + Buffer_Skip
      end do
    end if

    ! Add indentation and output.

    do Buffer_loc = 1, Buffer_Size
      Output_1 = Output_Buffer(Buffer_loc)
      Output_Buffer(Buffer_loc) = Blanks(1:A_Indent) // Output_1
    end do
    call Parallel_Write (Output_Buffer, A_Unit)

    ! Clean up.

    call Finalize (Output_Buffer)

    ! Format statements. With these formats, this should work up to
    ! (10^6 - 1) PEs.

100 format (/, 2a, /)
101 format (3a)
102 format (2a, l2)
103 format (2a, i12, :, 3(',', i12, :), a)
104 format (2a, '  ', 1p, e13.5e3)
```

```
105 format (2a, ' (', 1p, e13.5e3, ',', e13.5e3, ')')
106 format (2x, a, i5, a, i12, a, 1p, e13.5e3, :, &
          2(',', e13.5e3, :), ',', /, &
          (36x, e13.5e3, :, 2(',', e13.5e3, :), ','))
107 format (2x, a, i5, a, i11, a, i13, :, &
          2(',', i13, :), ',', /, &
          (36x, i13, :, 2(',', i13, :), ','))

    ! Verify guarantees - none.

    return
  end subroutine Output_ELL_Matrix
```

G.2.11 Read_Harwell_Boeing_ELL_Matrix Procedure

The main documentation of the Read_Harwell_Boeing_ELL_Matrix Procedure in § 12.2.11 on page 149 contains additional explanation of this code listing.

```
  subroutine Read_Harwell_Boeing_ELL_Matrix (ELLM, RHS_MV, Solution_MV, &
                                            Guess_MV, Row_Structure, &
                                            Column_Structure, Unit, status)

    ! Input variables.

    type(integer), intent(in), optional :: Unit       ! Input unit.

    ! Output variables.

    ! ELL_Matrix to be initialized.
    type(ELL_Matrix_type), intent(out) :: ELLM
    type(Status_type), intent(out), optional :: status  ! Exit status.
    ! Row and Column Base_Structures.
    type(Base_Structure_type), intent(inout), target :: Row_Structure, &
                                                       Column_Structure
    ! RHS mathematic vector (RHSVAL).
    type(Mathematic_Vector_type), optional :: RHS_MV
    ! Guess mathematic vector (SGUESS).
    type(Mathematic_Vector_type), optional :: Guess_MV
    ! Solution mathematic vector (XEXACT).
    type(Mathematic_Vector_type), optional :: Solution_MV

    ! Internal variables.

    type(integer) :: A_Unit                             ! Actual output unit.
    type(Status_type), dimension(33) :: allocate_status ! Allocation Status.
    type(Status_type) :: consolidated_status            ! Consolidated Status.
    type(integer) :: Max_Nonzeros     ! Maximum number of nonzero elements/row.
    type(integer) :: Max_Nonzeros_PE  ! Maximum number of nonzero elements/row,
                                      !   set to zero on non-IO PEs.

    ! Line 1:
    type(character,name_length) :: Name        ! Matrix name (TITLE).
```

```
type(character,8)  :: Key                  ! Key for matrix (KEY).

! Line 2: Number of lines in the file, for...
type(integer) :: NTotal_Lines              ! Total, not including
                                           !   header (TOTCRD).
type(integer) :: NEntries_of_Columns_Lines ! Entries of Columns
                                           !   (PTRCRD).
type(integer) :: NRows_of_Entries_Lines    ! Rows of Entries (INDCRD).
type(integer) :: NValues_Lines             ! Numerical values (VALCRD).
type(integer) :: NRHS_Lines                ! Right-hand sides,
                                           !   including starting
                                           !   guesses and solution
                                           !   vectors (RHSCRD).
! Line 3:
type(character,3)  :: Matrix_Type          ! Matrix type (MXTYPE).
type(integer) :: NRows                     ! Number of rows (NROW).
type(integer) :: NColumns                  ! Number of Columns (NCOL).
type(integer) :: NEntries                  ! Number of nonzero entries in
                                           !   the matrix (NNZERO).
type(integer) :: NElemental_Matrices       ! Number of elemental matrices
                                           !   (NELTVL).

! Line 4: Format strings for...
type(character,16) :: Entries_of_Columns_Format ! Entries of Columns
                                                !   (PTRFMT).
type(character,16) :: Rows_of_Entries_Format    ! Rows of Entries (INDFMT).
type(character,20) :: Values_Format             ! Values (VALFMT).
type(character,20) :: RHS_Format                ! RHS values (RHSFMT).

! Line 5:
type(character,3)  :: RHS_Type             ! RHS type (RHSTYP).
type(integer) :: NRHS                      ! Number of RHS vectors, not
                                           !   including solution or guess
                                           !   vectors (NRHS).
type(integer) :: NRHS_Nonzero_Rows         ! Number of RHS nonzero rows
                                           !   (NRHSIX).

! Data:
type(integer,1) :: Entries_of_Columns      ! Entry numbers of each column
                                           !   (POINTR).
type(integer,1) :: Rows_of_Entries         ! Row numbers of each Entry
                                           !   (ROWIND).
type(real,1) :: Values_CSC                 ! Numerical values for the
                                           !   Compressed Sparse Column
                                           !   (CSC) matrix (VALUES).
type(real,2) :: Values                     ! Numerical values for the
                                           !   ELL matrix (VALUES).
type(integer,2) :: Columns                 ! Column indices for the
                                           !   ELL matrix ().
type(real,2) :: Values_PE                  ! Numerical values for the
                                           !   ELL matrix on this PE
                                           !   (VALUES).
type(integer,2) :: Columns_PE              ! Column indices for the
                                           !   ELL matrix on this PE
```

```
                                          !   ().
  type(real,1) :: RHS_Values_PE             ! RHS vector for each PE
                                          !   (RHSVAL).
  type(real,1) :: Solution_PE               ! Solution vector for each PE
                                          !   (XEXACT).
  type(real,1) :: Guess_PE                  ! Initial guess vector for each
                                          !   PE (SGUESS).
  ! Temporary structures for input.
  type(real,1) :: Temporary_BNV
  type(Assembled_Vector_type) :: Temporary_AV
  type(Distributed_Vector_type) :: Temporary_DV

  ! Other internal variables.
  type(integer,1) :: Row_Length_Vector      ! Row lengths on all the PEs.
  type(integer,1) :: Column_Length_Vector   ! Column lengths on all the PEs.
  type(character,name_length) :: Locus_Name  ! Structure locus name.
  type(integer) :: column, column_PE, entry, row ! Loop parameters.
  type(integer,1) :: ELL_Column             ! Current ELL column number.
  type(integer,1) :: Nonzeros_per_Row       ! Number of nonzero entries
                                          !   per row.
  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  ! ELLM, Row_Structure, Column_Structure have not been initialized.
  VERIFY(.not.Initialized(ELLM),5)
  VERIFY(.not.Initialized(Row_Structure),5)
  VERIFY(.not.Initialized(Column_Structure),5)

  ! Set unit number.

  if (PRESENT(Unit)) then
    A_Unit = Unit
  else
    A_Unit = 5
  end if

  ! Allocations and initializations.

  call Initialize (allocate_status)
  call Initialize (consolidated_status)

  ! Read in the header block.

  if (this_is_IO_PE) then
    Name = ' '
    read (A_Unit,100) Name(1:72), Key
    read (A_Unit,101) NTotal_Lines, NEntries_of_Columns_Lines, &
                      NRows_of_Entries_Lines, NValues_Lines, NRHS_Lines
    read (A_Unit,102) Matrix_Type, NRows, NColumns, NEntries, &
                      NElemental_Matrices
    read (A_Unit,103) Entries_of_Columns_Format, Rows_of_Entries_Format,&
                      Values_Format, RHS_Format
    if (NRHS_Lines > 0) then
```

```
    read (A_Unit,104) RHS_Type, NRHS, NRHS_Nonzero_Rows
  else
    RHS_Type = 'F  '
    NRHS = 0
    NRHS_Nonzero_Rows = 0
  end if
end if

! Format statements.

100 format (a72, a8)
101 format (5i14)
102 format (a3, 11x, 4i14)
103 format (2a16, 2a20)
104 format ( a3, 11x, 2i14 )

! Broadcast needed information (could put this into a
! smaller number of broadcasts (by type) later).

call Broadcast (Matrix_Type)
call Broadcast (NColumns)
call Broadcast (NElemental_Matrices)
call Broadcast (NEntries)
call Broadcast (NRHS)
call Broadcast (NRHS_Lines)
call Broadcast (NRows)
call Broadcast (NValues_Lines)
call Broadcast (RHS_Type)

! Input checks.

! Can only read real, unsymmetric, assembled matrices.
VERIFY(Matrix_Type=='RUA' .or. Matrix_Type=='rua',2)
! Can only read assembled matrices (warn on this only because
! often-used SPARSKIT routines do this incorrectly).
WARN_IF(NElemental_Matrices/=0,2)
! Can read at most one right-hand-side vector.
VERIFY(NRHS<=1,2)
! Cannot read sparse-storage RHS vectors.
VERIFY(.not.(NRHS_Lines>0 .and. RHS_Type(1:1)/='F'),2)
! Gots to be a matrix.
VERIFY(NValues_Lines>0,2)

! Set Row and Column Structures (divide evenly among the PEs).

call Initialize (Row_Length_Vector, NPEs)
call Generate_Even_Distribution (Row_Length_Vector, NRows)
Locus_Name = 'Rows'
call Initialize (Row_Structure, Row_Length_Vector, Locus_Name, &
                 allocate_status(1))

call Initialize (Column_Length_Vector, NPEs)
call Generate_Even_Distribution (Column_Length_Vector, NColumns)
Locus_Name = 'Columns'
```

```
call Initialize (Column_Structure, Column_Length_Vector, Locus_Name, &
                 allocate_status(2))

! Read in matrix structure.

call Initialize (Entries_of_Columns, NColumns+1, allocate_status(3))
call Initialize (Rows_of_Entries, NEntries, allocate_status(4))
if (this_is_IO_PE) then
  read (A_Unit,Entries_of_Columns_Format) Entries_of_Columns
  read (A_Unit,Rows_of_Entries_Format) Rows_of_Entries
end if

! Determine Max_Nonzeros.

call Initialize (Nonzeros_per_Row, NRows, allocate_status(5))
if (this_is_IO_PE) then
  do entry = 1, NEntries
    Nonzeros_per_Row(Rows_of_Entries(entry)) = &
      Nonzeros_per_Row(Rows_of_Entries(entry)) + 1
  end do
  Max_Nonzeros = MAXVAL(Nonzeros_per_Row)
end if
call Broadcast (Max_Nonzeros)
call Finalize (Nonzeros_per_Row, allocate_status(6))

! Initialize temporary BNA ELL structure - on non-IO PEs, this is a
!                                          dummy, unused structure.

if (this_is_IO_PE) then
  Max_Nonzeros_PE = Max_Nonzeros
else
  Max_Nonzeros_PE = 1
end if
call Initialize (Values,  NRows, Max_Nonzeros_PE, allocate_status(7))
call Initialize (Columns, NRows, Max_Nonzeros_PE, allocate_status(8))

! Read matrix values.

call Initialize (Values_CSC, NEntries, allocate_status(9))
if (this_is_IO_PE) then
  read (A_Unit,Values_Format) Values_CSC
end if

! Convert to ELL structure.

call Initialize (ELL_Column, NRows, allocate_status(10))
if (this_is_IO_PE) then
  ELL_Column = 0
  do column = 1, NColumns
    do entry = Entries_of_Columns(column), Entries_of_Columns(column+1)-1
      row = Rows_of_Entries(entry)
      ELL_Column(row) = ELL_Column(row) + 1
      Values(row,ELL_Column(row)) = Values_CSC(entry)
      Columns(row,ELL_Column(row)) = column
```

```
      end do
    end do
  else
    ELL_Column = 1
  end if
  ! Max_Nonzeros matches ELL_Column.
  VERIFY(MAXVAL(ELL_Column)==Max_Nonzeros_PE,7)
  call Finalize (ELL_Column, allocate_status(11))
  call Finalize (Values_CSC, allocate_status(12))
  call Finalize (Entries_of_Columns, allocate_status(13))
  call Finalize (Rows_of_Entries, allocate_status(14))

  ! Distribute matrix to PEs.

  call Initialize (Values_PE,  Row_Length_Vector(this_PE), Max_Nonzeros, &
                   allocate_status(15))
  call Initialize (Columns_PE, Row_Length_Vector(this_PE), Max_Nonzeros, &
                   allocate_status(16))
  do column = 1, Max_Nonzeros
    if (this_is_IO_PE) then
      column_PE = column
    else
      column_PE = 1
    end if
    call Distribute (Values_PE(:,column),  Values(:,column_PE),  &
                     Row_Length_Vector)
    call Distribute (Columns_PE(:,column), Columns(:,column_PE), &
                     Row_Length_Vector)
  end do
  call Finalize (Values, allocate_status(17))
  call Finalize (Columns, allocate_status(18))

  ! Initialize matrix structure and set.

  call Initialize (ELLM, Max_Nonzeros, Row_Structure, &
                   Column_Structure, Name, allocate_status(19))
  call Set_Values (ELLM, Values_PE, Columns_PE)
  call Finalize (Values_PE, allocate_status(20))
  call Finalize (Columns_PE, allocate_status(21))

  ! Read Right-Hand Sides.

  if (NRHS > 0) then

    if (PRESENT(RHS_MV)) then
      call Initialize (Temporary_BNV, NRows, allocate_status(22))
      call Initialize (Temporary_AV, Row_Structure, 1, &
                       status=allocate_status(23))
      call Initialize (Temporary_DV, Row_Structure, 1, &
                       status=allocate_status(24))
      call Initialize (RHS_Values_PE, Row_Length_Vector(this_PE), &
                       status=allocate_status(25))
      call Initialize (RHS_MV, Row_Structure, 'RHS: '//Name, &
                       allocate_status(26))
```

```
      if (this_is_IO_PE) then
        read (A_Unit,RHS_Format) Temporary_BNV
      end if
      Temporary_AV = Temporary_BNV
      Temporary_DV = Temporary_AV
      RHS_Values_PE = Temporary_DV
      RHS_MV = RHS_Values_PE
    end if

    ! Read starting guesses.

    if (RHS_Type(2:2) == 'G' .and. PRESENT(Guess_MV)) then

      call Initialize (Guess_PE, Column_Length_Vector(this_PE), &
                       status=allocate_status(27))
      call Initialize (Guess_MV, Column_Structure, 'Guess: '//Name, &
                       allocate_status(28))
      if (this_is_IO_PE) then
        read (A_Unit,RHS_Format) Temporary_BNV
      end if
      Temporary_AV = Temporary_BNV
      Temporary_DV = Temporary_AV
      Guess_PE = Temporary_DV
      Guess_MV = Guess_PE

    end if

    ! Read solution vectors.

    if (RHS_Type(3:3) == 'X' .and. PRESENT(Solution_MV)) then

      call Initialize (Solution_PE, Column_Length_Vector(this_PE), &
                       status=allocate_status(29))
      call Initialize (Solution_MV, Column_Structure, 'Solution: '//Name, &
                       allocate_status(30))
      if (this_is_IO_PE) then
        read (A_Unit,RHS_Format) Temporary_BNV
      end if
      Temporary_AV = Temporary_BNV
      Temporary_DV = Temporary_AV
      Solution_PE = Temporary_DV
      Solution_MV = Solution_PE
    end if

    ! Clean up.

    call Finalize (Temporary_BNV, allocate_status(31))
    call Finalize (Temporary_AV, allocate_status(32))
    call Finalize (Temporary_DV, allocate_status(33))
  end if

  ! Process status variables.

  consolidated_status = allocate_status
```

```
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)     ! ELL_Matrix is now valid.

    return
  end subroutine Read_Harwell_Boeing_ELL_Matrix
```

G.2.12 Residual_ELL_Matrix Procedure

The main documentation of the Residual_ELL_Matrix Procedure in § 12.2.12 on page 150 contains additional explanation of this code listing.

```
  subroutine Residual_ELL_Matrix (Residual_MV, A_ELLM, X_MV, B_MV)

    ! Input variables.

    ! ELL Matrix to be multiplied.
    type(ELL_Matrix_type), intent(inout) :: A_ELLM
    ! Mathematic Vector to be multiplied.
    type(Mathematic_Vector_type), intent(inout) :: X_MV
    ! Mathematic Vector to be subtracted.
    type(Mathematic_Vector_type), intent(inout) :: B_MV

    ! Output variable.

    ! Residual vector.
    type(Mathematic_Vector_type), intent(inout) :: Residual_MV

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(A_ELLM),5)            ! A_ELLM is valid.
    VERIFY(Valid_State(X_MV),5)              ! X_MV is valid.
    VERIFY(Valid_State(B_MV),5)              ! B_MV is valid.
    VERIFY(Valid_State(Residual_MV),5)       ! Residual_MV is valid.
    ! X_MV has same Structure as A_ELLM Column_Structure.
    VERIFY(ASSOCIATED(A_ELLM%Column_Structure,X_MV%Structure),5)
    ! B_MV and Residual_MV have same Structure as A_ELLM Row_Structure.
    VERIFY(ASSOCIATED(A_ELLM%Row_Structure,B_MV%Structure),5)
    VERIFY(ASSOCIATED(A_ELLM%Row_Structure,Residual_MV%Structure),5)

    ! Do the MatVec, store in the Residual (Residual = A x).
```

```
    call MatVec (A_ELLM, X_MV, Residual_MV)

    ! Subtract B. (Residual = Residual - B).

    Residual_MV%Values = Residual_MV%Values - B_MV%Values

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (Residual_MV)

    ! Verify guarantees - none.

    return
  end subroutine Residual_ELL_Matrix
```

G.2.13 Set_0_Diagonal_to_1_ELL_Matrix Procedure

The main documentation of the Set_0_Diagonal_to_1_ELL_Matrix Procedure in § 12.2.13 on page 150 contains additional explanation of this code listing.

```
  subroutine Set_0_Diagonal_to_1_ELL_Matrix (ELLM)

    ! Use association information.

    use Caesar_Numbers_Module, only: zero, one

    ! Input variables - none.

    ! Input/Output variable.

    type(ELL_Matrix_type), intent(inout) :: ELLM ! Variable to be set.

    ! Internal variables.

    type(integer) :: Row      ! Row number.
    type(integer) :: shift    ! Index shift.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)                           ! ELLM is valid.

    ! Set the diagonal to unity if the equation is missing (row is all zero).

    shift = -First_PE(ELLM%Row_Structure) + 1
    do Row = 1, Length_PE(ELLM%Row_Structure)

      if (ALL(ELLM%Values(Row,:) == zero) .AND. &
          ALL(ELLM%Columns(Row,:) == 0)) then
        ELLM%Values(Row,1) = one
```

```
      ELLM%Columns(Row,1) = Row - shift   ! Must convert column number
    end if                                ! from local to global.

  end do

  ! Unset the updated? variables.

  call Set_Not_Up_to_Date (ELLM)

  ! Verify guarantees.

  VERIFY(Valid_State(ELLM),5)                          ! ELLM is still valid.

  return
end subroutine Set_0_Diagonal_to_1_ELL_Matrix
```

G.2.14 Set_Not_Up_to_Date_ELL_Matrix Procedure

The main documentation of the Set_Not_Up_to_Date_ELL_Matrix Procedure in § 12.2.14 on page 150 contains additional explanation of this code listing.

```
subroutine Set_Not_Up_to_Date_ELLM (ELLM)

  ! Input/Output variable.

  type(ELL_Matrix_type), intent(inout) :: ELLM ! Variable to be set.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(ELLM),5)                          ! ELLM is valid.

  ! Unset the updated? variables.

  ELLM%Average_is_Updated = .false.
  ELLM%Frobenius_Norm_is_Updated = .false.
  ELLM%Index_is_Updated = .false.
  ELLM%Infinity_Norm_is_Updated = .false.
  ELLM%Maximum_is_Updated = .false.
  ELLM%Minimum_is_Updated = .false.
  ELLM%One_Norm_is_Updated = .false.
  ELLM%Sum_is_Updated = .false.
  ELLM%Two_Norm_is_Updated = .false.

  ! Verify guarantees.

  VERIFY(Valid_State(ELLM),5)                          ! ELLM is still valid.

  return
end subroutine Set_Not_Up_to_Date_ELLM
```

G.2.15 Set_Values_ELL_Matrix Procedure

The main documentation of the Set_Values_ELL_Matrix Procedure in § 12.2.15 on page 151 contains additional explanation of this code listing.

```
subroutine Set_Values_ELL_Matrix_0 (ELLM, Value, Row, Column, Global)

  ! Note: this routine is very similar to Add_Values_ELL_Matrix_0.

  ! Input variables.

  type(real), intent(in) :: Value               ! Value scalar.
  type(integer), intent(in) :: Row              ! Row integer scalar.
  type(integer), intent(in) :: Column           ! Column integer scalar.
  type(logical), intent(in), optional :: Global ! Global/local index toggle.

  ! Input/Output variable.

  type(ELL_Matrix_type), intent(inout) :: ELLM  ! Variable to be set.

  ! Internal variables.

  type(logical) :: A_Global        ! Actual global/local toggle.
  type(integer) :: Column_Location ! Location in Columns array for the entry.
  type(integer) :: location        ! Loop index.
  type(integer) :: shift           ! Index shift.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(ELLM),5)                       ! ELLM is valid.
  VERIFY(Valid_State(Value),5)                      ! Value is valid.
  VERIFY(Valid_State(Row),5)                        ! Row is valid.
  VERIFY(Valid_State(Column),5)                     ! Column is valid.

  ! Global/Local toggle.

  if (PRESENT(Global)) then
    A_Global = Global
  else
    A_Global = .true.
  end if
  if (A_Global) then
    shift = -First_PE(ELLM%Row_Structure) + 1
  else
    shift = 0
  end if

  ! Another requirement check -- require that Row be in the correct range.

  VERIFY(Row + shift .InInterval. (/1, Length_PE(ELLM%Row_Structure)/),5)
```

```
    ! Set the value.

    if (Row /= 0) then

      ! Find a column location to store the entry.

      Column_Location = 0
      do location = 1, ELLM%Max_Nonzeros
        if (ELLM%Columns(Row + shift, location) == 0 .or. &
            ELLM%Columns(Row + shift, location) == Column) then
          Column_Location = location
          exit
        end if
      end do

      ! Store the entry.

      ELLM%Values(Row + shift, Column_Location) = Value
      ELLM%Columns(Row + shift, Column_Location) = Column

    else

      ! Make sure Column_Location has a non-zero value if Row=0,
      ! so the check below executes correctly.
      Column_Location = -1

    end if

    ! Make sure Max_Nonzeros is not exceeded (that is, that we
    ! found a spot to put the entry).

    VERIFY(Column_Location /= 0,1)

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (ELLM)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                    ! ELLM is still valid.

    return
  end subroutine Set_Values_ELL_Matrix_0

  subroutine Set_Values_ELL_Matrix_1 (ELLM, Values, Rows, Columns, Global)

    ! Note: this routine is very similar to Add_Values_ELL_Matrix_1.

    ! Input variables.

    type(real,1,np), intent(in) :: Values          ! Values bare naked array.
    type(integer,1,np), intent(in) :: Rows         ! Rows integer vector.
    type(integer,1,np), intent(in) :: Columns      ! Columns integer vector.
    type(logical), intent(in), optional :: Global  ! Global/local index toggle.
```

```
! Input/Output variable.

type(ELL_Matrix_type), intent(inout) :: ELLM ! Variable to be set.

! Internal variables.

type(logical) :: A_Global                   ! Actual global/local toggle.
type(integer) :: Column_Location            ! Location in Columns array
                                            !   for the entry.
type(integer) :: i                          ! Loop parameter.
type(integer) :: location                   ! Loop index.
type(logical) :: Max_Nonzeros_Not_Exceeded  ! Toggle for error check.
type(integer) :: shift                      ! Index shift.
define([rows_are_set_twice_check], [9])
ifelse(ARCHITECTURE, Sun, [], [
  ifelse(m4_eval(DEBUG_LEVEL >= rows_are_set_twice_check), 1, [
    type(integer) :: j                      ! Verify loop parameter.
  ])
])

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(ELLM),5)                         ! ELLM is valid.
VERIFY(Valid_State_NP(Values),5)                    ! Values is valid.
VERIFY(Valid_State_NP(Rows),5)                      ! Rows is valid.
VERIFY(Valid_State_NP(Columns),5)                   ! Columns is valid.
! Values, Rows & Columns size checks.
VERIFY(SIZE(Values,1) <= Length_PE(ELLM%Row_Structure),5)
VERIFY(SIZE(Rows) <= Length_PE(ELLM%Row_Structure),5)
VERIFY(SIZE(Rows) == SIZE(Values,1),5)
VERIFY(SIZE(Values) == SIZE(Columns),5)

! Global/Local toggle.

if (PRESENT(Global)) then
  A_Global = Global
else
  A_Global = .true.
end if
if (A_Global) then
  shift = -First_PE(ELLM%Row_Structure) + 1
else
  shift = 0
end if

! More requirement checks -- require that Rows entries are in the
! correct range.

VERIFY(Rows + shift .InInterval. (/1, Length_PE(ELLM%Row_Structure)/),5)

! Set the values.
```

```
  Max_Nonzeros_Not_Exceeded = .true.
  do i = 1, SIZE(Rows)
    if (Rows(i) /= 0) then

      ! Find a column location to store the entry.

      Column_Location = 0
      do location = 1, ELLM%Max_Nonzeros
        if (ELLM%Columns(Rows(i) + shift, location) == 0 .or. &
            ELLM%Columns(Rows(i) + shift, location) == Columns(i)) then
          Column_Location = location
          exit
        end if
      end do

      ! Make sure Max_Nonzeros is not exceeded (that is, that we
      ! found a spot to put the entry).

      Max_Nonzeros_Not_Exceeded = &
        Max_Nonzeros_Not_Exceeded .and. Column_Location /= 0

      ! Store the entry.

      ELLM%Values(Rows(i) + shift, Column_Location) = Values(i)
      ELLM%Columns(Rows(i) + shift, Column_Location) = Columns(i)

    end if
  end do

  ! Make sure Max_Nonzeros is not exceeded (that is, that we
  ! found a spot to put all of the entries).

  VERIFY(Max_Nonzeros_Not_Exceeded,1)

  ! Make sure no rows are set twice.

  ! This check bombed on Suns for NPES=16 or 32,
  ! so it has been removed there.
  ifelse(ARCHITECTURE, Sun, [], [
    VERIFY((/(((Rows(i)/=Rows(j) .and. Rows(i)/=0), dnl
      j=i+1,SIZE(Rows)), i=1,SIZE(Rows))/),rows_are_set_twice_check)
  ])

  ! Unset the updated? variables.

  call Set_Not_Up_to_Date (ELLM)

  ! Verify guarantees.

  VERIFY(Valid_State(ELLM),5)                    ! ELLM is still valid.

  return
end subroutine Set_Values_ELL_Matrix_1
```

```
subroutine Set_Values_ELL_Matrix_2 (ELLM, Values, Rows, Columns, Global)

  ! Note: this routine is very similar to Add_Values_ELL_Matrix_2.

  ! Input variables.

  type(real,2,np), intent(in) :: Values          ! Values bare naked array.
  type(integer,1,np), intent(in) :: Rows         ! Rows integer vector.
  type(integer,2,np), intent(in) :: Columns      ! Columns integer array.
  type(logical), intent(in), optional :: Global  ! Global/local index toggle.

  ! Input/Output variable.

  type(ELL_Matrix_type), intent(inout) :: ELLM ! Variable to be set.

  ! Internal variables.

  type(logical) :: A_Global                   ! Actual global/local toggle.
  type(integer) :: Column_Location            ! Location in Columns array
                                              !   for the entry.
  type(integer) :: i, j                       ! Loop parameters.
  type(integer) :: location                   ! Loop index.
  type(logical) :: Max_Nonzeros_Not_Exceeded  ! Toggle for error check.
  type(integer) :: shift                      ! Index shift.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(ELLM),5)                         ! ELLM is valid.
  VERIFY(Valid_State_NP(Values),5)                    ! Values is valid.
  VERIFY(Valid_State_NP(Rows),5)                      ! Rows is valid.
  VERIFY(Valid_State_NP(Columns),5)                   ! Columns is valid.
  ! Values, Rows & Columns size checks.
  VERIFY(SIZE(Values,1) <= Length_PE(ELLM%Row_Structure),5)
  VERIFY(SIZE(Values,2) == ELLM%Max_Nonzeros,5)
  VERIFY(SIZE(Rows) <= Length_PE(ELLM%Row_Structure),5)
  VERIFY(SIZE(Rows) == SIZE(Values,1),5)
  VERIFY(SIZE(Values) == SIZE(Columns),5)

  ! Global/Local toggle.

  if (PRESENT(Global)) then
    A_Global = Global
  else
    A_Global = .true.
  end if
  if (A_Global) then
    shift = -First_PE(ELLM%Row_Structure) + 1
  else
    shift = 0
  end if
```

```
! More requirement checks -- require that Rows entries are in the
! correct range.

VERIFY(Rows + shift .InInterval. (/1, Length_PE(ELLM%Row_Structure)/),5)

! Set the values.

Max_Nonzeros_Not_Exceeded = .true.
do i = 1, SIZE(Rows)
  if (Rows(i) /= 0) then
    do j = 1, SIZE(Values,2)

      ! Find a column location to store the entry.

      Column_Location = 0
      do location = 1, ELLM%Max_Nonzeros
        if (ELLM%Columns(Rows(i) + shift, location) == 0 .or. &
            ELLM%Columns(Rows(i) + shift, location) == Columns(i,j)) then
          Column_Location = location
          exit
        end if
      end do

      ! Make sure Max_Nonzeros is not exceeded (that is, that we
      ! found a spot to put the entry).

      Max_Nonzeros_Not_Exceeded = &
        Max_Nonzeros_Not_Exceeded .and. Column_Location /= 0

      ! Store the entry.

      ELLM%Values(Rows(i) + shift, Column_Location) = Values(i,j)
      ELLM%Columns(Rows(i) + shift, Column_Location) = Columns(i,j)

    end do
  end if
end do

! Make sure Max_Nonzeros is not exceeded (that is, that we
! found a spot to put all of the entries).

VERIFY(Max_Nonzeros_Not_Exceeded,1)

! Make sure no rows are set twice.

! This check bombed on Suns for NPES=16 or 32,
! so it has been removed there.
ifelse(ARCHITECTURE, Sun, [], [
  VERIFY((/(((Rows(i)/=Rows(j) .and. Rows(i)/=0), dnl
    j=i+1,SIZE(Rows)), i=1,SIZE(Rows))/),9)
])

! Make sure no columns are set twice.
```

```
    ! Next line was too long for the Absoft compiler,
    ! even with all spaces removed.

    !VERIFY((/((((Columns(i,j)/=Columns(i,k) .and. Columns(i,j)/=0), dnl
    !  k=j+1,SIZE(Columns,2)), j=1,SIZE(Columns,2)), i=1,SIZE(Columns,1))/),9)

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (ELLM)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                    ! ELLM is still valid.

    return
  end subroutine Set_Values_ELL_Matrix_2

  subroutine Set_Values_ELL_Matrix_All (ELLM, Values, Columns)

    ! Input variables.

    type(real,2,np), intent(in) :: Values         ! Values bare naked array.
    type(integer,2,np), intent(in) :: Columns     ! Columns bare naked array.

    ! Input/Output variable.

    type(ELL_Matrix_type), intent(inout) :: ELLM ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)                          ! ELLM is valid.
    VERIFY(Valid_State_NP(Values),5)                     ! Values is valid.
    VERIFY(Valid_State_NP(Columns),5)                    ! Columns is valid.
    ! Values and Columns size checks.
    VERIFY(SIZE(Values,1) == Length_PE(ELLM%Row_Structure),5)
    VERIFY(SIZE(Values,2) == ELLM%Max_Nonzeros,5)
    VERIFY(SIZE(Values) == SIZE(Columns),5)

    ! Set the values.

    ELLM%Values = Values
    ELLM%Columns = Columns

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (ELLM)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                          ! ELLM is still valid.

    return
```

```
  end subroutine Set_Values_ELL_Matrix_All

  subroutine Set_Value_ELL_Matrix_All (ELLM, Value)

    ! Input variables.

    type(real), intent(in) :: Value              ! Value scalar.

    ! Input/Output variable.

    type(ELL_Matrix_type), intent(inout) :: ELLM ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)                          ! ELLM is valid.
    VERIFY(Valid_State(Value),5)                         ! Value is valid.

    ! Set all the values which have nonzero
    ! column indices to the input scalar.

    where (ELLM%Columns /= 0)
      ELLM%Values = Value
    end where

    ! Unset the updated? variables.

    call Set_Not_Up_to_Date (ELLM)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)                          ! ELLM is still valid.

    return
  end subroutine Set_Value_ELL_Matrix_All
```

G.2.16 ELL_Matrix Class Unit Test Program

This lightly commented program performs a unit test on the ELL_Matrix Class, which is described in § 12.2 on page 142.

```
program Unit_Test

  use Caesar_Data_Structures_Module
  use Caesar_Mathematic_Vector_Class
  use Caesar_ELL_Matrix_Class
  use Caesar_Numbers_Module, only: zero, half, one, two, three, six, ten
  implicit none

  type(Communication_type) :: Comm
  type(Base_Structure_type) :: Row_Structure, Column_Structure
```

```
type(Status_type) :: status
type(character,name_length) :: Locus_Name, ELLM_Name, MV_Name
type(integer,1) :: Row_Length_Vector, Rows
type(integer,2) :: Columns
type(integer) :: shift, first_plus_last_PE_RS
type(real,2) :: Values
type(real,1) :: X_BNV
type(real) :: AvgB, B_Norm2, MaxB, MinB, NRows_real, ResidualNorm1, &
              ResidualNorm2
type(integer) :: i, j, NRows, MaxNonzeros
type(ELL_Matrix_type) :: A_ELLM, B_ELLM
type(Mathematic_Vector_type) :: X_MV, B_MV, Residual_MV

! Initializations.

call Initialize (Comm)
call Output (Comm)
call Initialize (status)
call Initialize (Locus_Name)
call Initialize (ELLM_Name)
call Initialize (MV_Name)
call Initialize (Row_Length_Vector, NPEs)
Row_Length_Vector = (/ (i**2, i = 1, NPEs) /)
call Initialize (MaxNonzeros)
MaxNonzeros = MIN(5, SUM(Row_Length_Vector))

! Local temp vectors.

call Initialize (Rows, Row_Length_Vector(this_PE))
call Initialize (Values,  Row_Length_Vector(this_PE), MaxNonzeros)
call Initialize (Columns, Row_Length_Vector(this_PE), MaxNonzeros)
call Initialize (X_BNV, Row_Length_Vector(this_PE))

! Initialize base structure, ELL matrices and mathematic vectors.

Locus_Name = 'Rows'
call Initialize (Row_Structure, Row_Length_Vector, Locus_Name, status)
Locus_Name = 'Columns'
call Initialize (Column_Structure, Row_Length_Vector, Locus_Name, status)
ELLM_Name = 'Coefficients'
call Initialize (A_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 ELLM_Name, status)
call Initialize (B_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 ELLM_Name, status)
MV_Name = 'Variables'
call Initialize (X_MV, Column_Structure, MV_Name, status)
MV_Name = 'Equations'
call Initialize (B_MV, Row_Structure, MV_Name, status)
NRows = Length_Total(Row_Structure)

! Testing statements.

! Note that there can be no procedure calls inside a loop which
! is executed a different number of times on each PE, because
```

```
! global communication is done for verifications within a procedure.

shift = - First_PE(Row_Structure) + 1
first_plus_last_PE_RS = Last_PE(Row_Structure) + First_PE(Row_Structure)
do i = First_PE(Row_Structure), Last_PE(Row_Structure)
  ! Values(i,:) = global i
  Values(i + shift,:) = changetype(real, i) / MaxNonzeros
  ! Rows(i) = global i, but reversed in order on each PE
  Rows(i + shift) =  first_plus_last_PE_RS - i
  ! Columns(i,:) = max((global i)-4,1) + (0,1,2,3,4)
  do j = 1, MaxNonzeros
    Columns(i + shift,j) = MAX(i-4,1) + j-1
  end do
end do
VERIFY(Max_Nonzeros(A_ELLM)==MaxNonzeros,2)

! Set A_ELLM all at once:
!
!   1 PE:   A_ELLM = [ 1 ]
!   >1 PEs: A_ELLM = [ 0.2 0.2 0.2 0.2 0.2                     ]
!                    [ 0.4 0.4 0.4 0.4 0.4                     ]
!                    [ 0.6 0.6 0.6 0.6 0.6                     ]
!                    [ 0.8 0.8 0.8 0.8 0.8                     ]
!                    [ 1.0 1.0 1.0 1.0 1.0                     ]
!                    [     1.2 1.2 1.2 1.2 1.2                 ]
!                    [         1.4 1.4 1.4 1.4 1.4             ]
!                    [             1.6 1.6 1.6 1.6 1.6         ]
!                    [                 1.8 1.8 1.8 1.8 1.8     ]
!                    [                     2.0 2.0 2.0 2.0 2.0 ]
!      and matrix N = sum(i**2, i = 1, NPEs)

call Set_Values (A_ELLM, Values, Columns)

! Set B_ELLM via 'partial replacement' call (even though we are
! replacing the whole thing) with half of Values, then use the
! Add_Values procedure to add the other half.
! Note that B_ELLM values are in reverse order on each PE.

Values = Values/2.d0
call Set_Values (B_ELLM, Values, Rows, Columns)
call Add_Values (B_ELLM, Values, Rows, Columns)

! Output the ELLMs.

call Output (A_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Output (B_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! MatVec test.

X_BNV = one                   ! Stuff X with 1's.
X_MV = X_BNV
call MatVec (A_ELLM, X_MV, B_MV)
call Output (B_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
MaxB = Maximum(B_MV)
```

```
  MinB = Minimum(B_MV)
  AvgB = Average(B_MV)
  if (.not. VeryClose(MinB, one) ) then
    if (this_is_IO_PE) then
      write (6,*) 'Error in Minimum B => ', MinB
    end if
  end if
  if (.not. VeryClose(MaxB, changetype(real,NRows)) ) then
    if (this_is_IO_PE) then
      write (6,*) 'Error in Maximum B => ', MaxB
    end if
  end if
  if (.not. VeryClose(AvgB, (MaxB+MinB)*half) ) then
    if (this_is_IO_PE) then
      write (6,*) 'Error in Average B => ', AvgB
    end if
  end if

  ! This should use already-communicated values to get the same answer.

  call MatVec (A_ELLM, X_MV, B_MV)
  call Output (B_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Norm test for resultant of MatVec (which should be {1,2,3,4,...}).
  ! This makes use of the fact that
  !
  !  --
  !  \     2     1     3      2
  !  /    k   =  - (2 n  + 3 n  + n)
  !  --          6
  ! k=1,n

  B_Norm2 = Two_Norm(B_MV)
  NRows_real = changetype(real,NRows)
  if (this_is_IO_PE) then
    write (6,100) 'Norm test:'
    write (6,100) '  ||B||_2 = ', B_Norm2
100 format (/,a,f15.1)
    if (.not. VeryClose(B_Norm2**2, &
        (two * NRows_real**3 + three * NRows_real**2 + NRows_real)/six)) then
      write (6,*) 'Error -- ||B||_2 is incorrect.'
    end if
  end if

  ! This MatVec should require new communication.

  X_BNV = ten                     ! Stuff X with 10's.
  X_MV = X_BNV
  call MatVec (A_ELLM, X_MV, B_MV)
  call Output (B_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  if (.not. VeryClose(Average(B_MV), AvgB*ten) ) then
    if (this_is_IO_PE) then
      write (6,*) 'Error -- Two consecutive MatVecs give incorrect answers.'
    end if
```

```
end if

! Check some things that are known:

if (.not. VeryClose(Average(A_ELLM), &
    changetype(real,half*(NRows+1)/NRows)) ) then
  if (this_is_IO_PE) then
    write (6,*) 'Error -- Average is incorrect.'
  end if
end if
if (.not. Sum(A_ELLM) >= One_Norm(A_ELLM)) then
  if (this_is_IO_PE) then
    write (6,*) 'Error -- Sum or One_Norm is incorrect.'
  end if
end if
if (.not. Sum(A_ELLM) >= Infinity_Norm(A_ELLM)) then
  if (this_is_IO_PE) then
    write (6,*) 'Error -- Sum or Infinity_Norm is incorrect.'
  end if
end if
if (.not. VeryClose(Minimum(A_ELLM), &
    one/changetype(real,Max_Nonzeros(A_ELLM)))) then
  if (this_is_IO_PE) then
    write (6,*) 'Error -- Minimum is incorrect.'
  end if
end if

! Tweak one value of A_ELLM on each PE and re-output.

call Set_Values (A_ELLM, 10.d0, Last_PE(Row_Structure), &
                 Last_PE(Row_Structure))
call Output (A_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check state of the ELL matrices.

VERIFY(Valid_State(A_ELLM),0)
VERIFY(Valid_State(B_ELLM),0)

! Test the Harwell-Boeing reader.

call Finalize (X_MV)   ! Note: must finalize X_MV before A_ELLM because
                       ! the MatVec with A_ELLM created an OV in X_MV
                       ! based on the Data_Index in A_ELLM.
call Finalize (B_MV)
call Finalize (A_ELLM)
call Finalize (B_ELLM)
call Finalize (Row_Structure)
call Finalize (Column_Structure)
if (this_is_IO_PE) then
  open (unit=8, &
    File='source/class/linear_algebra/battery/Augustus_Prob0_MH_K2_8x8x8.hb')
end if
call Read_Harwell_Boeing (A_ELLM, RHS_MV=B_MV, Solution_MV=X_MV, &
                          Row_Structure=Row_Structure, &
```

```
                           Column_Structure=Column_Structure, Unit=8, &
                           status=status)
  call Output (A_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  call Output (X_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  call Output (B_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Test Residual and MatVec.

  call Initialize (Residual_MV, Row_Structure, 'Residual', status)
  call Residual (Residual_MV, A_ELLM, X_MV, B_MV)
  ResidualNorm1 = Norm(Residual_MV)

  call MatVec (A_ELLM, X_MV, B_MV)
  if (this_is_IO_PE) then
    write (6,101) 'The next mathematic vector should be the same as', &
                  'the previous one, except for roundoff.'
101 format (/,a,/,a)
  end if
  call Output (B_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  call Residual (Residual_MV, A_ELLM, X_MV, B_MV)
  ResidualNorm2 = Norm(Residual_MV)

  if (this_is_IO_PE) then
    ! Correct value for ResidualNorm1 is:
    !   8.145846809344343E-13 on Intel/Absoft
    !   8.145895568345390E-13 on Intel/Lahey
    if (ResidualNorm1 < 8.14d-13 .or. &
        ResidualNorm1 > 8.15d-13) then
      write (6,*) 'Residual Norm 1 out of bounds: ', ResidualNorm1
    end if
    if (.not. VeryClose(ResidualNorm2,zero)) then
      write (6,*) 'Error: Residual Norm 2 is nonzero: ', ResidualNorm2
    end if
  end if

  ! Also need
  ! 1: call Add (ELLM1, ELLM2, ELLM3) or call Add_to_ELLM (ELLM1, ELLM2)
  ! 2: ELLM1 = ELLM2
  ! 3: call Scale (ELLM, scalar)
  ! 4: ELLM1 == ELLM2

  ! Check state of the ELL matrices.

  VERIFY(Valid_State(A_ELLM),0)

  ! Finalize objects and communications.

  call Finalize (X_MV)   ! Note: must finalize X_MV before A_ELLM because
                         ! the MatVec with A_ELLM created an OV in X_MV
                         ! based on the Data_Index in A_ELLM.
  call Finalize (A_ELLM)
  call Finalize (Rows)
  call Finalize (Values)
```

```
  call Finalize (Columns)
  call Finalize (X_BNV)
  call Finalize (B_MV)
  call Finalize (Row_Structure)
  call Finalize (Column_Structure)
  call Finalize (Comm)

end
```

G.3 Solver Class Code Listing

The main documentation of the Solver Class in § 12.3 on page 152 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 03/08/04
! CVS Info:   $Id: solver.F90,v 1.11 2008/09/23 00:23:59 hall Exp $

module Caesar_Solver_Class

  ! Global use associations.

  use Caesar_Data_Structures_Module
  use Caesar_Mathematic_Vector_Class
  use Caesar_ELL_Matrix_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Set, Solve
  ifdef([USE_LAMG],[
    public :: Convert
  ])

  interface Initialize
    module procedure Initialize_Solver
  end interface

  interface Finalize
    module procedure Finalize_Solver
  end interface
```

```
  interface Valid_State
    module procedure Valid_State_Solver
  end interface

  interface Initialized
    module procedure Initialized_Solver
  end interface

  ifdef([USE_LAMG],[
    interface Convert
      module procedure Convert_ELL_to_LAMG
    end interface
  ])

  interface Set
    module procedure Set_Solver_Variable
  end interface

  interface Solve
    module procedure Solve
  end interface

  ! Public type definitions.

  public :: Solver_type

  type Solver_type

    ! Initialization flag.

    type(integer) :: Initialized

    ! Package name.

    type(character,name_length) :: Package

    ! Non-Package-Dependent info.

    type(character,name_length) :: Stopping_Test  ! Test used to signify
                                                  ! convergence.
    type(real) :: Epsilon                         ! Error tolerance.
    type(integer) :: Maximum_Iterations           ! Maximum iteration count
                                                  ! allowed.

    ! LAMG package information.

    type(integer) :: LAMG_levout  ! Package output: 0=silent, 4=verbose.

  end type Solver_type

contains
```

The Solver Class contains the following routines which are listed in separate sections:

end module Caesar_Solver_Class

G.3.1 Initialize_Solver Procedure

The main documentation of the Initialize_Solver Procedure in § 12.3.1 on page 153 contains additional explanation of this code listing.

```
  subroutine Initialize_Solver (Solver, Package, status)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variables.

    type(character,*), intent(in) :: Package          ! Solver package to call.

    ! Output variables.

    type(Solver_type), intent(out) :: Solver        ! Solver to be initialized.
    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(1) :: allocate_status  ! Allocation Status.
    type(Status_type) :: consolidated_status            ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(.not.Valid_State(Solver),5)    ! Solver is not valid.

    ! Allocations and initializations.

    call Initialize (allocate_status)
    call Initialize (consolidated_status)
```

```
    ! Set up internals.

    if (Package(1:4) == 'LAMG') then
      Solver%Package = 'LAMG'
    else
      ! No Package match found.
      VERIFY(.false., 0)
    end if

    ! Default Solver settings.

    Solver%Epsilon = 1.d-10
    Solver%Stopping_Test = '||r||/||b|| < Eps'
    Solver%Maximum_Iterations = 1000

    ! Default LAMG settings.

    if (Solver%Package == 'LAMG') then

      Solver%LAMG_levout = 2     ! Package output: 0=silent, 4=verbose.

    end if

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set initialization flag.

    Solver%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(Solver),5)  ! Solver is now valid.

    return
  end subroutine Initialize_Solver
```

G.3.2 Finalize_Solver Procedure

The main documentation of the Finalize_Solver Procedure in § 12.3.2 on page 153 contains additional explanation of this code listing.

```
subroutine Finalize_Solver (Solver, status)

  ! Use associations.

  use Caesar_Flags_Module, only: uninitialized_flag

  ! Input/Output variable.

  ! Solver to be finalized.
  type(Solver_type), intent(inout) :: Solver

  ! Output variables.

  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(Status_type), dimension(5) :: deallocate_status ! Deallocation Status.
  type(Status_type) :: consolidated_status             ! Consolidated Status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Solver),7) ! Solver is valid.

  ! Unset initialization flag.

  Solver%Initialized = uninitialized_flag

  ! Deallocations and finalizations.

  ! Set deallocation status.

  call Initialize (deallocate_status)
  call Initialize (consolidated_status)

  ! Finalize internals.

  call Finalize (Solver%Epsilon,            deallocate_status(1))
  call Finalize (Solver%LAMG_levout,        deallocate_status(2))
  call Finalize (Solver%Maximum_Iterations, deallocate_status(3))
  call Finalize (Solver%Package,            deallocate_status(4))
  call Finalize (Solver%Stopping_Test,      deallocate_status(5))

  ! Process status variables.

  consolidated_status = deallocate_status
  if (PRESENT(status)) then
    WARN_IF(Error(consolidated_status), 5)
    status = consolidated_status
  else
    VERIFY(Normal(consolidated_status), 5)
  end if
```

```
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(Solver),7) ! Solver is not valid.

    return
  end subroutine Finalize_Solver
```

G.3.3 Valid_State_Solver Procedure

The main documentation of the Valid_State_Solver Procedure in § 12.3.3 on page 154 contains additional explanation of this code listing.

```
  function Valid_State_Solver (Solver) result(Valid)

    ! Use association information.

    use Caesar_Numbers_Module, only: zero

    ! Input variables.

    ! Variable to be checked.
    type(Solver_type), intent(in) :: Solver

    ! Output variables.

    type(logical) :: Valid             ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for validity of internals.

    Valid = Valid .and. Initialized(Solver)
    Valid = Valid .and. Valid_State(Solver%Epsilon)
    Valid = Valid .and. Valid_State(Solver%LAMG_levout)
    Valid = Valid .and. Valid_State(Solver%Maximum_Iterations)
    Valid = Valid .and. Valid_State(Solver%Stopping_Test)
    if (.not.Valid) return

    ! Checks on the validity of Solver.

    Valid = Valid .and. (Solver%Package == 'LAMG')
          ! .or. Solver%Package == 'Other Package'
    Valid = Valid .and. Solver%Epsilon >= zero
    Valid = Valid .and. Solver%Maximum_Iterations >= 0
    Valid = Valid .and. (Solver%Stopping_Test == '||r||/||b|| < Eps')
```

```
        ! .or. Solver%Stopping_Test == 'Other Test'

    return
  end function Valid_State_Solver
```

G.3.4 Initialized_Solver Procedure

The main documentation of the Initialized_Solver Procedure in § 12.3.4 on page 154 contains additional explanation of this code listing.

```
  function Initialized_Solver (Solver) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    type(Solver_type), intent(in) :: Solver  ! Solver to be checked.

    ! Output variable.

    type(logical) :: Initialized             ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = Solver%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Solver
```

G.3.5 Set_Solver_Variable Procedure

The main documentation of the Set_Solver_Variable Procedure in § 12.3.5 on page 154 contains additional explanation of this code listing.

```
  subroutine Set_Solver_Variable (Solver, Variable, Real_Value, &
                                  Integer_Value, Character_Value, &
                                  Logical_Value)

    ! Input variables.

    type(character,*), intent(in) :: Variable ! Variable name.
```

```
    ! Variable value.
    type(character,*), intent(in), optional :: Character_Value
    type(real), intent(in), optional :: Real_Value
    type(integer), intent(in), optional :: Integer_Value
    type(logical), intent(in), optional :: Logical_Value

    ! Input/Output variables.

    type(Solver_type), intent(inout) :: Solver   ! Solver to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Solver),5)  ! Solver is valid.

    ! Set Solver variables.

    if (Variable == 'Epsilon' .and. PRESENT(Real_Value)) then
      Solver%Epsilon = Real_Value

    else if (Variable == 'Maximum_Iterations' .and. &
             PRESENT(Integer_Value)) then
      Solver%Maximum_Iterations = Integer_Value

    else if (Variable == 'Stopping_Test' .and. PRESENT(Character_Value)) then
      Solver%Stopping_Test = Character_Value

    ! Set LAMG variables.

    else if (Variable == 'LAMG_levout' .and. PRESENT(Integer_Value)) then
      Solver%LAMG_levout = Integer_Value

    ! Error check.

    else
      ! No variable match found.
      VERIFY(.false.,0)
    end if

    ! Verify guarantees.

    VERIFY(Valid_State(Solver),5)  ! Solver is valid.

    return
  end subroutine Set_Solver_Variable
```

G.3.6 Convert_ELL_to_LAMG Procedure

The main documentation of the Convert_ELL_to_LAMG Procedure in § 12.3.6 on page 155 contains additional explanation of this code listing.

```
ifdef([USE_LAMG],[
  subroutine Convert_ELL_to_LAMG (LAMG_Matrix, ELLM, LAMG_Communicator, &
                                  LAMG_Options, status)

    ! Use association information.

    use Caesar_Numbers_Module, only: zero
    use LAMG_Module

    ! Input variables.

    type(ELL_Matrix_type), intent(in) :: ELLM         ! Matrix in ELL format.
    type(lamg_comm), intent(in) :: LAMG_Communicator  ! Communicator.
    type(LAMG_LS_Options), intent(in) :: LAMG_Options ! LAMG options.

    ! Output variables.

    type(LAMG_Matrix_dcsr_r), intent(out) :: LAMG_Matrix ! LAMG formt matrix.
    type(status_type), intent(out),optional :: status ! Output status.

    ! Internal variables.

    type(integer) :: NNonzeros_PE         ! Number of nonzeros on this PE.
    type(integer) :: LAMG_Status          ! LAMG status.
    type(integer,2) :: Columns_BNA        ! Matrix columns bare naked array.
    type(real,2) :: Values_BNA            ! Matrix values bare naked array.
    type(LAMG_Matrix_csr_r) :: LAMG_BR_Matrix ! LAMG block-row format matrix.
    type(integer) :: BR_Location          ! Location in the block-row matrix.
    type(integer) :: ELL_Location         ! Location in the ELL matrix.
    type(integer) :: row                  ! Row loop counter.

    type(Status_type), dimension(10) :: allocate_status ! Allocation Status.
    type(Status_type) :: consolidated_status           ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(ELLM),5)  ! ELLM is valid.

    ! Initializations.

    call Initialize (allocate_status)
    call Initialize (consolidated_status)

    ! Pull ELLM matrix values and columns out in BNAs.

    call Initialize (Values_BNA,  NRows_PE(ELLM), Max_Nonzeros(ELLM), &
                     allocate_status(1))
    call Initialize (Columns_BNA, NRows_PE(ELLM), Max_Nonzeros(ELLM), &
                     allocate_status(2))
    Values_BNA = ELLM
    Columns_BNA = ELLM
    NNonzeros_PE = COUNT(Values_BNA/=zero)
```

```
! Convert ELLM into LAMG matrix.

LAMG_BR_Matrix%nrow = NRows_PE(ELLM)
LAMG_BR_Matrix%ncol = NRows_Total(ELLM)
LAMG_BR_Matrix%nnz  = NNonzeros_PE

! Initialize LAMG block-row matrix.

call Initialize (LAMG_BR_Matrix%ia, LAMG_BR_Matrix%nrow+1, &
                 allocate_status(3))
call Initialize (LAMG_BR_Matrix%ja, LAMG_BR_Matrix%nnz   , &
                 allocate_status(4))
call Initialize (LAMG_BR_Matrix% a, LAMG_BR_Matrix%nnz   , &
                 allocate_status(5))

! Fill the LAMG block-row matrix.

BR_location = 0
do row = 1, NRows_PE(ELLM)
  LAMG_BR_Matrix%ia(row) = BR_location + 1
  do ELL_location = 1, Max_Nonzeros(ELLM)
    if (Values_BNA(row,ELL_location) /= zero) then
      BR_location = BR_location + 1
      LAMG_BR_Matrix% a(BR_location) =  Values_BNA(row,ELL_location)
      LAMG_BR_Matrix%ja(BR_location) = Columns_BNA(row,ELL_location)
    end if
  end do
end do
LAMG_BR_Matrix%ia(NRows_PE(ELLM) + 1) = BR_location + 1
VERIFY(BR_location==NNonzeros_PE,4)

! Finalize BNA matrix.

call Finalize (Values_BNA, allocate_status(6))
call Finalize (Columns_BNA, allocate_status(7))

! Transform matrix into LAMG internal format.

call LAMG_Transform_Matrix (LAMG_Matrix, LAMG_BR_Matrix, &
                            LAMG_Communicator, LAMG_Options, LAMG_Status)

! Finalize LAMG block-row matrix.

call Finalize (LAMG_BR_Matrix%ia, allocate_status(8))
call Finalize (LAMG_BR_Matrix%ja, allocate_status(9))
call Finalize (LAMG_BR_Matrix% a, allocate_status(10))

! Process status variables.

consolidated_status = allocate_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
```

```
      else
        VERIFY(Normal(consolidated_status), 5)
      end if
      call Finalize (consolidated_status)
      call Finalize (allocate_status)

      ! Verify guarantees - none.

      return
    end subroutine Convert_ELL_to_LAMG
  ])
```

G.3.7 Solve Procedure

The main documentation of the Solve Procedure in § 12.3.7 on page 156 contains additional explanation of this code listing.

```
  define([RESIDUAL_DEBUG_LEVEL],[6])

  subroutine Solve (Solver, A_ELLM, X_MV, B_MV)

    ! Use association information.

    ifdef([USE_LAMG],[
      use LAMG_Module
    ])
    ifdef([USE_MPI],[
      include(mpif.h)
    ])

    ! Input variables.

    type(Solver_type), intent(in) :: Solver             ! Solver to be set.
    type(ELL_Matrix_type), intent(inout) :: A_ELLM      ! Matrix in ELL format.
    type(Mathematic_Vector_type), intent(inout) :: B_MV ! RHS vector.

    ! Input/Output variables.


    ! Output variables.

    type(Mathematic_Vector_type), intent(inout) :: X_MV ! Solution vector.

    ! Internal variables.

    type(Status_type) :: status

    ! LAMG variables:
    ifdef([USE_LAMG],[
      type(LAMG_comm) :: LAMG_Communicator  ! LAMG communicator.
      type(LAMG_LS_Options) :: LAMG_Options ! LAMG options.
      type(integer) :: LAMG_Status          ! LAMG status.
```

```
  type(LAMG_Matrix_dcsr_r) :: A_LAMG    ! Matrix in LAMG format.
  type(real,1) :: X_LAMG                ! Solution vector in LAMG format.
  type(real,1) :: B_LAMG                ! RHS vector in LAMG format.
])
ifelse(m4_eval(DEBUG_LEVEL >= RESIDUAL_DEBUG_LEVEL), 1, [
  type(Mathematic_Vector_type) :: Residual_MV  ! Residual vector.
])

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

!VERIFY(Valid_State(input),0)  ! Quick, important test.
!VERIFY(Valid_State(input),9)  ! Slow, unimportant test.

! LAMG solve.

if (Solver%Package == 'LAMG') then

  ifdef([USE_LAMG],[

    ! Initialize LAMG communication.

    ifdef([USE_MPI],[
      call LAMG_Comm_Init (MPI_COMM_WORLD, LAMG_Communicator, LAMG_Status)
    ],[
      call LAMG_Comm_Init (             0, LAMG_Communicator, LAMG_Status)
    ])

    ! Initialize LAMG options.

    call LAMG_Options_Init (LAMG_Communicator, LAMG_Options, LAMG_Status)

    ! Extract RHS and Solution (Initial Guess) vector.

    call Initialize (B_LAMG, NRows_PE(A_ELLM), status)
    B_LAMG = B_MV
    call Initialize (X_LAMG, NRows_PE(A_ELLM), status)
    X_LAMG = X_MV

    ! Convert matrix to LAMG format.

    call Convert (A_LAMG, A_ELLM, LAMG_Communicator, LAMG_Options, status)

    ! Set LAMG options.

    ! Maximum allowed number of iterations.
    call LAMG_Set_Option_i (LAMG_itsmax, Solver%Maximum_Iterations, &
                            LAMG_Communicator, LAMG_Options, LAMG_Status)
    ! Convergence criterion.
    call LAMG_Set_Option_r (LAMG_tol, Solver%Epsilon, LAMG_Communicator, &
                            LAMG_Options, LAMG_Status)
    ! Package output: 0=silent, 4=verbose.
    call LAMG_Set_Option_i (LAMG_levout, Solver%LAMG_levout, &
```

```
                              LAMG_Communicator, LAMG_Options, LAMG_Status)
      ! LAMG does not currently allow any stopping tests except this one.
      VERIFY(Solver%Stopping_Test=='||r||/||b|| < Eps',5)

      ! Solve the linear system.

      call LAMG_Solve (A_LAMG, B_LAMG, X_LAMG, LAMG_Communicator, &
                       LAMG_Options, LAMG_Status)

      ! Set Solution vector in MV format.

      X_MV = X_LAMG

      ! Deallocate LAMG vectors and matrix.

      call Finalize (B_LAMG)
      call Finalize (X_LAMG)
      call LAMG_Free_dcsr_r (A_LAMG, LAMG_Communicator, LAMG_Options, &
                             LAMG_Status)

      ! Terminate LAMG options.

      call LAMG_Options_Term (LAMG_Communicator, LAMG_Options, LAMG_Status)

      ! Terminate LAMG communication.

      call LAMG_Comm_Term (LAMG_Communicator, LAMG_Status)

    ],[

      ! No LAMG available.

      write (6,*) '*********************************************'
      write (6,*) 'Error: The LAMG solver has been requested but'
      write (6,*) 'this executable was not compiled with LAMG.'
      write (6,*) '*********************************************'
      Call Abort ()

    ])

  ! Error check.

  else
    ! No variable match found.
    VERIFY(.false.,0)
  end if

  ! Verify guarantees.

  ifelse(m4_eval(DEBUG_LEVEL >= RESIDUAL_DEBUG_LEVEL), 1, [
    call Duplicate (Residual_MV, X_MV)
    call Residual (Residual_MV, A_ELLM, X_MV, B_MV)
    if (Solver%Stopping_Test=='||r||/||b|| < Eps') then
      VERIFY(Norm(Residual_MV)/Norm(B_MV)<=Solver%Epsilon,dnl
```

```
              RESIDUAL_DEBUG_LEVEL)
      end if
      call Finalize (Residual_MV)
    ])

    return
  end subroutine Solve
```

G.3.8 Solver Class Unit Test Program

This lightly commented program performs a unit test on the Solver Class, which is described in § 12.3 on page 152.

```
program Unit_Test

  use Caesar_Data_Structures_Module
  use Caesar_Mathematic_Vector_Class
  use Caesar_ELL_Matrix_Class
  use Caesar_Solver_Class
  implicit none

  type(Communication_type) :: Comm
  type(Base_Structure_type) :: Row_Structure, Column_Structure
  type(Status_type) :: status
  type(character,name_length) :: MV_Name
  type(integer) :: NRows
  type(ELL_Matrix_type) :: A_ELLM
  type(Mathematic_Vector_type) :: X_MV, B_MV, Residual_MV, Solution_MV
  type(Solver_type) :: Solver

  ! Initializations.

  call Initialize (Comm)
  call Output (Comm)
  call Initialize (status)
  call Initialize (Solver, 'LAMG')

  ! Check state of the Solver.

  VERIFY(Valid_State(Solver),0)

  ! Read in a linear system (this initializes and sets A_ELLM,
  !                          B_MV, X_MV, Row_Structure and
  !                          Column_Structure).

  if (this_is_IO_PE) then
    open (unit=8, &
      File='source/class/linear_algebra/battery/Augustus_Prob0_MH_K2_8x8x8.hb')
  end if
  call Read_Harwell_Boeing (A_ELLM, RHS_MV=B_MV, Solution_MV=X_MV, &
                           Row_Structure=Row_Structure, &
                           Column_Structure=Column_Structure, Unit=8, &
```

```
                            status=status)
  NRows = NRows_Total(A_ELLM)
  call Output (A_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  call Output (X_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  call Output (B_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Initializations.

  MV_Name = 'Calculated Solution'
  call Initialize (Solution_MV, Column_Structure, MV_Name)
  MV_Name = 'Calculated Residual'
  call Initialize (Residual_MV, Row_Structure, MV_Name)

  ! Solve the system.

  call Solve (Solver, A_ELLM, Solution_MV, B_MV)
  call Output (Solution_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Calculate residual.

  call Residual (Residual_MV, A_ELLM, Solution_MV, B_MV)
  call Output (Residual_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Check state of the Solver.

  VERIFY(Valid_State(Solver),0)

  ! Finalize objects and communications.

  call Finalize (Solver)
  call Finalize (A_ELLM)
  call Finalize (X_MV)
  call Finalize (B_MV)
  call Finalize (Row_Structure)
  call Finalize (Column_Structure)
  call Finalize (Solution_MV)
  call Finalize (Residual_MV)
  call Finalize (Comm)

end
```

Appendix H

Equation Module Code Listing

The main documentation of the Equation Module in chapter 13 on page 157 contains additional explanation of this code listing.

H.1 Monomial Class Code Listing

The main documentation of the Monomial Class in § 13.1 on page 157 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 01/31/05
! CVS Info:   $Id: monomial.F90,v 1.25 2009/09/07 23:27:25 hall Exp $

module Caesar_Monomial_Class

  ! Global use associations.

  use Caesar_Linear_Algebra_Module
  use Caesar_Multi_Mesh_Class

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Add_to_Matrix_Equation, Locus, Name, Output

  interface Initialize
    module procedure Initialize_Monomial
```

```
  end interface

  interface Finalize
    module procedure Finalize_Monomial
  end interface

  interface Valid_State
    module procedure Valid_State_Monomial
  end interface

  interface Initialized
    module procedure Initialized_Monomial
  end interface

  interface Add_to_Matrix_Equation
    module procedure Add_to_Matrix_Equation_Monomial
  end interface

  interface Locus
    module procedure Get_Locus_Monomial
  end interface

  interface Name
    module procedure Get_Name_Monomial
  end interface

  interface Output
    module procedure Output_Monomial
  end interface

  ! Public type definitions.

  public :: Monomial_type

  type Monomial_type

    ! Initialization flag.

    type(integer) :: Initialized

    ! The name for this variable.

    type(character,name_length) :: Name

    ! The coefficient of the monomial.

    type(real,1) :: Coefficient

    ! The degree of the monomial.

    type(real) :: Exponent

    ! The independent variable at the linearization value (previous iterate).
```

```
    type(real,1) :: Phi_k

    ! The independent variable at the past time step, only used for time
    ! derivative monomials.

    type(real,1) :: Phi_n

    ! Logical value which is true for time derivative monomials.

    type(logical) :: Derivative

    ! The time step size, only used for time derivative monomials.

    type(real) :: Delta_t

    ! Mesh that this monomial is defined on.

    type(Multi_Mesh_type), pointer :: Mesh

    ! Evaluation locus. (to be used in a future version -- for now,
    !                    assumed to be "Cells".)

    type(character,name_length) :: Locus

    ! Locus Base_Structure.

    type(Base_Structure_type), pointer :: Structure

    ! Variable number, equation number and total number of equations for
    ! this monomial.

    type(integer) :: Variable, Equation, NEquations

  end type Monomial_type

contains
```

The Monomial Class contains the following routines which are listed in separate sections:

Initialize_Monomial (§ H.1.1, page 634)

Finalize_Monomial (§ H.1.2, page 636)

Valid_State_Monomial (§ H.1.3, page 638)

Initialized_Monomial (§ H.1.4, page 639)

Add_to_Matrix_Equation_Monomial (§ H.1.5, page 640)

Get Value Monomial (§ H.1.6, page 642)

Output_Monomial (§ H.1.7, page 643)

end module Caesar_Monomial_Class

H.1.1 Initialize_Monomial Procedure

The main documentation of the Initialize_Monomial Procedure in § 13.1.1 on page 158 contains additional explanation of this code listing.

```
subroutine Initialize_Monomial (Monomial, Coefficient, Exponent, Phi_k_MV, &
                                Locus, Mesh, Name, Derivative, Delta_t, &
                                Phi_n_MV, Variable, Equation, NEquations, &
                                status)

  ! Use associations.

  use Caesar_Flags_Module,   only: initialized_flag
  use Caesar_Numbers_Module, only: one

  ! Input variables.

  type(real), intent(in) :: Exponent                ! Monomial degree.
  type(real,1) :: Coefficient                       ! Monomial coefficient.
  type(character,*), intent(in) :: Locus            ! Evaluation locus.
  type(Multi_Mesh_type), target :: Mesh             ! Monomial Mesh.
  ! Previous iterate value of the independent variable.
  type(Mathematic_Vector_type), intent(inout) :: Phi_k_MV
  ! Past time step value of the independent variable.
  type(Mathematic_Vector_type), intent(inout), optional :: Phi_n_MV
  type(character,*), intent(in), optional :: Name   ! Monomial name.
  type(logical), intent(in), optional :: Derivative ! Time derivative toggle.
  type(real), intent(in), optional :: Delta_t       ! Time step size.
  type(integer), intent(in), optional :: Variable   ! Variable number.
  type(integer), intent(in), optional :: Equation   ! Equation number.
  type(integer), intent(in), optional :: NEquations ! Number of equations.

  ! Output variables.

  type(Monomial_type), intent(out) :: Monomial ! Monomial to be initialized.
  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(Status_type), dimension(3) :: allocate_status  ! Allocation Status.
  type(Status_type) :: consolidated_status            ! Consolidated Status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(.not.Valid_State(Monomial),5)  ! Monomial is not valid.
  VERIFY(Valid_State(Coefficient),5)    ! Coefficient is valid.
  VERIFY(Valid_State(Exponent),5)       ! Exponent is valid.

  ! Set up pointers.

  Monomial%Mesh => Mesh
```

```
    select case (Locus)
    case ("Cells")
      Monomial%Structure => Cell_Structure(Mesh)
    case ("Nodes")
      Monomial%Structure => Node_Structure(Mesh)
    case ("Faces")
      Monomial%Structure => Face_Structure(Mesh)
    end select

    ! Set time derivative toggle.

    if (PRESENT(Derivative)) then
      Monomial%Derivative = Derivative
    else
      Monomial%Derivative = .false.
    end if

    ! Three more requirements: Coefficient, Phi_k_MV and Phi_n_MV are the
    ! right size.

    VERIFY(SIZE(Coefficient)==Length_PE(Monomial%Structure),5)
    VERIFY(Length_PE(Phi_k_MV)==Length_PE(Monomial%Structure),5)
    if (PRESENT(Phi_n_MV)) then
      VERIFY(Length_PE(Phi_n_MV)==Length_PE(Monomial%Structure),5)
    end if

    ! Allocations and initializations.

    call Initialize (allocate_status)
    call Initialize (consolidated_status)
    call Initialize (Monomial%Exponent, allocate_status(1))
    call Initialize (Monomial%Coefficient, Length_PE(Monomial%Structure), &
                     allocate_status(2))
    call Initialize (Monomial%Phi_k, Length_PE(Monomial%Structure), &
                     allocate_status(3))
    if (Monomial%Derivative) then
      call Initialize (Monomial%Phi_n, Length_PE(Monomial%Structure), &
                       allocate_status(3))
    end if

    ! Set up internals.

    if (PRESENT(Name)) Monomial%Name = Name
    Monomial%Coefficient = Coefficient
    Monomial%Exponent    = Exponent
    Monomial%Locus       = Locus
    Monomial%Phi_k       = Phi_k_MV
    if (Monomial%Derivative) then
      if (PRESENT(Phi_n_MV)) then
        Monomial%Phi_n   = Phi_n_MV
      else
        Monomial%Phi_n   = Phi_k_MV
      end if
    end if
```

```
    if (PRESENT(Delta_t)) then
      Monomial%Delta_t = Delta_t
    else
      Monomial%Delta_t = one
    end if
    if (PRESENT(Equation)) then
      Monomial%Equation = Equation
    else
      Monomial%Equation = 1
    end if
    if (PRESENT(Variable)) then
      Monomial%Variable = Variable
    else
      ! If not specified, use the variable for this equation.
      Monomial%Variable = Monomial%Equation
    end if
    if (PRESENT(NEquations)) then
      Monomial%NEquations = NEquations
    else
      Monomial%NEquations = 1
    end if

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set initialization flag.

    Monomial%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(Monomial),5)  ! Monomial is now valid.

    return
  end subroutine Initialize_Monomial
```

H.1.2 Finalize_Monomial Procedure

The main documentation of the Finalize_Monomial Procedure in § 13.1.2 on page 158 contains additional explanation of this code listing.

```
  subroutine Finalize_Monomial (Monomial, status)
```

```
! Use associations.

use Caesar_Flags_Module, only: uninitialized_flag

! Input/Output variable.

! Monomial to be finalized.
type(Monomial_type), intent(inout) :: Monomial

! Output variables.

type(Status_type), intent(out), optional :: status  ! Exit status.

! Internal variables.

type(Status_type), dimension(10) :: deallocate_status !Deallocation Status.
type(Status_type) :: consolidated_status             ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Monomial),7) ! Monomial is valid.

! Unset initialization flag.

Monomial%Initialized = uninitialized_flag

! Deallocations and finalizations.

! Set deallocation status.

call Initialize (deallocate_status)
call Initialize (consolidated_status)

! Finalize internals.

NULLIFY(Monomial%Mesh)
NULLIFY(Monomial%Structure)
call Finalize (Monomial%Exponent,    deallocate_status(1))
call Finalize (Monomial%Coefficient, deallocate_status(2))
call Finalize (Monomial%Phi_k,       deallocate_status(3))
call Finalize (Monomial%Locus,       deallocate_status(4))
call Finalize (Monomial%Name,        deallocate_status(5))
if (Monomial%Derivative) then
  call Finalize (Monomial%Phi_n,     deallocate_status(6))
end if
call Finalize (Monomial%Delta_t,     deallocate_status(7))
call Finalize (Monomial%Variable,    deallocate_status(8))
call Finalize (Monomial%Equation,    deallocate_status(9))
call Finalize (Monomial%NEquations,  deallocate_status(10))

! Process status variables.
```

```
    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(Monomial),7) ! Monomial is not valid.

    return
  end subroutine Finalize_Monomial
```

H.1.3 Valid_State_Monomial Procedure

The main documentation of the Valid_State_Monomial Procedure in § 13.1.3 on page 159 contains additional explanation of this code listing.

```
  function Valid_State_Monomial (Monomial) result(Valid)

    ! Input variables.

    ! Variable to be checked.
    type(Monomial_type), intent(in) :: Monomial

    ! Output variables.

    type(logical) :: Valid             ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for association of pointered internals.

    Valid = Valid .and. ASSOCIATED(Monomial%Mesh)
    Valid = Valid .and. ASSOCIATED(Monomial%Structure)
    if (.not.Valid) return

    ! Check for validity of internals.

    Valid = Valid .and. Initialized(Monomial)
    Valid = Valid .and. Valid_State(Monomial%Coefficient)
    Valid = Valid .and. Valid_State(Monomial%Delta_t)
    Valid = Valid .and. Valid_State(Monomial%Equation)
    Valid = Valid .and. Valid_State(Monomial%Exponent)
```

```
    Valid = Valid .and. Valid_State(Monomial%Locus)
    Valid = Valid .and. Valid_State(Monomial%Name)
    Valid = Valid .and. Valid_State(Monomial%NEquations)
    Valid = Valid .and. Valid_State(Monomial%Phi_k)
    if (Monomial%Derivative) then
      Valid = Valid .and. Valid_State(Monomial%Phi_n)
    end if
    Valid = Valid .and. Valid_State(Monomial%Variable)
    if (.not.Valid) return

    ! Checks on the validity of Monomial.

    Valid = Valid .and. &
            (Monomial%Equation .InInterval. (/ 1, Monomial%NEquations /))

    return
  end function Valid_State_Monomial
```

H.1.4 Initialized_Monomial Procedure

The main documentation of the Initialized_Monomial Procedure in § 13.1.4 on page 159 contains additional explanation of this code listing.

```
  function Initialized_Monomial (Monomial) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    type(Monomial_type), intent(in) :: Monomial  ! Monomial to be checked.

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = Monomial%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Monomial
```

H.1.5 Add_to_Matrix_Equation_Monomial Procedure

The main documentation of the Add_to_Matrix_Equation_Monomial Procedure in § 13.1.5 on page 160 contains additional explanation of this code listing.

```
subroutine Add_to_Matrix_Equation_Monomial (Monomial, ELLM, RHS_MV)

  ! Use association information.

  use Caesar_Numbers_Module, only: one, zero

  ! Input variables.

  type(Monomial_type), intent(inout) :: Monomial  ! Monomial to be added.

  ! Input/Output variables.

  ! ELL_Matrix to be incremented.
  type(ELL_Matrix_type), intent(inout) :: ELLM
  ! RHS mathematic vector.
  type(Mathematic_Vector_type), intent(inout) :: RHS_MV

  ! Internal variables.

  ! Arrays to manipulate values of the matrix.
  type(real,1) :: Matrix_Values
  type(integer,1) :: Matrix_Rows
  type(integer,1) :: Matrix_Columns
  type(integer) :: i                          ! Loop parameter.
  type(integer) :: shift                      ! Index shift.
  type(real,1) :: Volume_Cells                ! Volume of the cells.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Monomial),5) ! Monomial is valid.
  VERIFY(Valid_State(ELLM),5)     ! ELLM is valid.
  VERIFY(Valid_State(RHS_MV),5)   ! RHS_MV is valid.

  ! Create some working vectors.

  call Initialize (Matrix_Values, NCells_PE(Monomial%Mesh))
  call Initialize (Matrix_Rows, NCells_PE(Monomial%Mesh))
  call Initialize (Matrix_Columns, NCells_PE(Monomial%Mesh))
  call Initialize (Volume_Cells, NCells_PE(Monomial%Mesh))

  ! Set Matrix_Rows, Matrix_Columns.
  !
  ! To spread out the rows and columns from this equation into a multiple
  ! equation matrix, translate local row numbers via:
  !   i --> NEquations*(i - 1) + Equation
  ! and global column numbers via:
```

```
!   i + shift --> NEquations*(i + shift - 1) + Variable

shift = First_PE(Monomial%Structure) - 1
do i = 1, Length_PE(Monomial%Structure)
  Matrix_Rows(i) = Monomial%NEquations*(i - 1) + Monomial%Equation
  Matrix_Columns(i) = Monomial%NEquations*(i + shift - 1) + &
                      Monomial%Variable
end do

! Increment ELLM values.

call Get_Volume_Cells (Volume_Cells, Monomial%Mesh)
if (Monomial%Exponent /= zero) then
  if (Monomial%Derivative) then
    ! Time derivative version.
    Matrix_Values(:) = Monomial%Coefficient * Monomial%Exponent * &
                       Monomial%Phi_k ** (Monomial%Exponent - one) * &
                       Volume_Cells / Monomial%Delta_t
  else
    ! Standard version.
    Matrix_Values(:) = Monomial%Coefficient * Monomial%Exponent * &
                       Monomial%Phi_k ** (Monomial%Exponent - one) * &
                       Volume_Cells
  end if
else
  ! Treat exponent=0 (monomial=constant) case separately to avoid
  ! division by zero if Phi_k=0 and to save a little time.
  Matrix_Values(:) = zero
end if
call Add_Values (ELLM, Matrix_Values, Matrix_Rows, Matrix_Columns, &
                 Global=.false.)

! Increment RHS_MV values. Note that they have been multiplied by -1
! to move them to the RHS of the equation.

if (Monomial%Derivative) then
  ! Time derivative version.
  Matrix_Values(:) = Monomial%Coefficient *  &
                     ((Monomial%Exponent - one) * &
                     Monomial%Phi_k ** Monomial%Exponent + &
                     Monomial%Phi_n ** Monomial%Exponent) * &
                     Volume_Cells / Monomial%Delta_t
else
  ! Standard version.
  Matrix_Values(:) = Monomial%Coefficient * &
                     (Monomial%Exponent - one) * &
                     Monomial%Phi_k ** Monomial%Exponent * &
                     Volume_Cells
end if
call Add_Values (RHS_MV, Matrix_Values, Matrix_Rows, Global=.false.)

! Finalize working vectors.

call Finalize (Matrix_Values)
```

```
    call Finalize (Matrix_Rows)
    call Finalize (Matrix_Columns)
    call Finalize (Volume_Cells)

    ! Verify guarantees.

    VERIFY(Valid_State(ELLM),5)     ! ELLM is valid.
    VERIFY(Valid_State(RHS_MV),5)   ! RHS_MV is valid.

    return
  end subroutine Add_to_Matrix_Equation_Monomial
```

H.1.6 Get Value Monomial Functions

The main documentation of the Get Value Monomial Functions in § 13.1.6 on page 161 contains additional explanation of this code listing.

```
  define([CHARACTER_ACCESS_ROUTINE],[
    pushdef([VALUE], [$1])
    pushdef([VALUE_Result], expand(VALUE_Result))
    pushdef([Get_CHARACTER_VALUE_Monomial], expand(Get_VALUE_Monomial))

    function Get_CHARACTER_VALUE_Monomial (Monomial) result(VALUE_Result)

      ! Input variables.

      type(Monomial_type), intent(in) :: Monomial   ! Monomial object.

      ! Output variables.

      type(character,80) :: VALUE_Result         ! Monomial value to be output.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(Monomial),5)               ! Monomial is valid.

      ! Set value.

      VALUE_Result = Monomial%Value

      ! Verify guarantees - none.

      return
    end function Get_CHARACTER_VALUE_Monomial

    popdef([VALUE])
    popdef([VALUE_Result])
    popdef([Get_CHARACTER_VALUE_Monomial])
  ])
```

```
fortext([Value],
        [Locus Name],[
  CHARACTER_ACCESS_ROUTINE(Value)
])
```

H.1.7 Output_Monomial Procedure

The main documentation of the Output_Monomial Procedure in § 13.1.7 on page 161 contains additional explanation of this code listing.

```
subroutine Output_Monomial (Monomial, First, Last, Unit, Indent, status)

  ! Input variables.

  ! Variable to be output.
  type(Monomial_type), intent(inout) :: Monomial
  type(integer), intent(in), optional :: First    ! Extents of value data
  type(integer), intent(in), optional :: Last     !   to be output.
  type(integer), intent(in), optional :: Unit     ! Output unit.
  type(integer), optional :: Indent               ! Indentation.

  ! Output variable.

  type(Status_type), intent(out), optional :: status  ! Exit status.

  ! Internal variables.

  type(integer) :: A_First                        ! Actual first value.
  type(integer) :: A_Last                         ! Actual last value.
  type(integer) :: A_Unit                         ! Actual output unit.
  type(integer) :: A_Indent                       ! Actual indentation.
  type(character,80) :: Blanks                    ! A line of blanks.
  type(Mathematic_Vector_type) :: Coefficient_MV  ! Temp MV for Coefficient.
  type(Mathematic_Vector_type) :: Phi_MV          ! Temp MV for Phi_k, Phi_n.
  type(Status_type), dimension(2) :: allocate_status ! Allocation Status.
  type(Status_type) :: consolidated_status           ! Consolidated Status.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Monomial),5)                  ! Monomial is valid.

  ! Set unit number.

  if (PRESENT(Unit)) then
    A_Unit = Unit
  else
    A_Unit = 6
  end if

  ! Set indentation.
```

```
if (PRESENT(Indent)) then
  A_Indent = Indent
else
  A_Indent = 0
end if
Blanks = ' '

! Set up local limits in terms of global limits.

if (PRESENT(First)) then
  A_First = First
else
  A_First = 1
end if
if (PRESENT(Last)) then
  A_Last = Last
else
  A_Last = Length_Total(Monomial%Structure)
end if

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)

! Output Identification Info.

if (this_is_IO_PE) then
  write (A_Unit,100) Blanks(1:A_Indent), 'Monomial Information:'
  write (A_Unit,101) Blanks(1:A_Indent+2), 'Name                = ', &
                     TRIM(Monomial%Name)
  write (A_Unit,102) Blanks(1:A_Indent+2), 'Initialized         =', &
                     Initialized(Monomial)
  write (A_Unit,103) Blanks(1:A_Indent+2), 'Exponent            =', &
                     Monomial%Exponent
  write (A_Unit,101) Blanks(1:A_Indent+2), 'Locus               = ', &
                     TRIM(Monomial%Locus)
  write (A_Unit,102) Blanks(1:A_Indent+2), 'Derivative          =', &
                     Monomial%Derivative
  write (A_Unit,103) Blanks(1:A_Indent+2), 'Delta t             =', &
                     Monomial%Delta_t
  write (A_Unit,104) Blanks(1:A_Indent+2), 'Variable            =', &
                     Monomial%Variable
  write (A_Unit,104) Blanks(1:A_Indent+2), 'Equation            =', &
                     Monomial%Equation
  write (A_Unit,104) Blanks(1:A_Indent+2), 'NEquations          =', &
                     Monomial%NEquations
end if

! Output internal structure info.

call Output (Monomial%Structure, A_Unit, 'Monomial Locus', A_Indent+2)
```

```
    call Initialize (Coefficient_MV, Monomial%Structure, &
                     'Coefficient as an MV', allocate_status(1))
    Coefficient_MV = Monomial%Coefficient
    call Output (Coefficient_MV, A_First, A_Last, A_Unit, A_Indent+2)
    call Finalize (Coefficient_MV)

    call Initialize (Phi_MV, Monomial%Structure, &
                     'Phi_k as an MV', allocate_status(2))
    Phi_MV = Monomial%Phi_k
    call Output (Phi_MV, A_First, A_Last, A_Unit, A_Indent+2)
    call Finalize (Phi_MV)

    if (Monomial%Derivative) then
      call Initialize (Phi_MV, Monomial%Structure, &
                       'Phi_n as an MV', allocate_status(2))
      Phi_MV = Monomial%Phi_n
      call Output (Phi_MV, A_First, A_Last, A_Unit, A_Indent+2)
      call Finalize (Phi_MV)
    end if

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Format statements.

    100 format (/, 2a, /)
    101 format (3a)
    102 format (2a, l2)
    103 format (2a, 1p, e13.5e3)
    104 format (2a, 1p, i4)

    ! Verify guarantees - none.

    return
  end subroutine Output_Monomial
```

H.1.8 Monomial Class Unit Test Program

This lightly commented program performs a unit test on the Monomial Class, which is described in § 13.1 on page 157.

```
program Unit_Test
```

```
use Caesar_Linear_Algebra_Module
use Caesar_Multi_Mesh_Class
use Caesar_Monomial_Class
use Caesar_Numbers_Module, only: zero, one, two, three
implicit none

type(Communication_type) :: Comm
type(Base_Structure_type) :: Row_Structure, Column_Structure
type(Status_type) :: status
type(real,1) :: Coefficient
type(real) :: Delta_t, Exponent
type(logical) :: Derivative
type(integer) :: i, NDimensions, NRows, MaxNonzeros, &
                 NCells_X, NCells_Y, NCells_Z, &
                 Equation, NEquations, Variable
type(real,1) :: Lengths
type(ELL_Matrix_type) :: A_ELLM
type(Mathematic_Vector_type) :: X_MV, B_MV, Phi_MV
type(Multi_Mesh_type) :: Mesh
type(Monomial_type) :: Constant_Term, Linear_Term, Quadratic_Term

! Initializations.

call Initialize (Comm)
call Output (Comm)
call Initialize (status)
MaxNonzeros = 3  ! Only need one, but why not test 3?

! Create a mesh.

NCells_X = 14
NCells_Y = 13
NCells_Z = 12
NDimensions = 3
call Initialize (Lengths, NDimensions)
Lengths = (/ 10.d0, 20.d0, 40.d0 /)
call Initialize (Mesh, NDimensions, Lengths, NCells_X, NCells_Y, NCells_Z, &
                 'Uniform Mesh', status)
Delta_t = zero
Derivative = .false.
Variable = 3
Equation = 3
NEquations = 3

! Initialize ELL Matrix and Mathematic Vectors.

call Generate_Multiple (Row_Structure, NEquations, Cell_Structure(Mesh), &
                        'Rows')
call Generate_Multiple (Column_Structure, NEquations, Cell_Structure(Mesh), &
                        'Columns')
call Initialize (A_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 'Coefficients', status)
call Initialize (X_MV, Column_Structure, 'Variables', status)
call Initialize (B_MV, Row_Structure, 'Equations', status)
```

```
NRows = Length_Total(Row_Structure)

! Initialize coefficent array and MV for past time step values (Phi).

call Initialize (Coefficient, Length_PE(Cell_Structure(Mesh)))
call Initialize (Phi_MV, Cell_Structure(Mesh), 'Phi')

! Set Phi_MV to two, using Coefficient as a temporary.

Coefficient = two
Phi_MV = Coefficient

! Initialize Constant Term.

Exponent = zero
Coefficient = (/ (three*i, i = First_Cell_PE(Mesh), &
                              Last_Cell_PE(Mesh)) /)
call Initialize (Constant_Term, Coefficient, Exponent, Phi_MV, 'Cells', &
                 Mesh, 'Source Term', Derivative, Delta_t, Phi_MV, &
                 Variable, Equation, NEquations, status)

! Initialize Linear Term.

Exponent = one
call Initialize (Linear_Term, Coefficient, Exponent, Phi_MV, 'Cells', &
                 Mesh, 'Removal Term', Derivative, Delta_t, Phi_MV, &
                 Variable, Equation, NEquations, status)

! Initialize Quadratic Term.

Exponent = two
call Initialize (Quadratic_Term, Coefficient, Exponent, Phi_MV, 'Cells', &
                 Mesh, 'Quadratic Term', Derivative, Delta_t, Phi_MV, &
                 Variable, Equation, NEquations, status)

! Output Terms.

call Output ( Constant_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Output (   Linear_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Output (Quadratic_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check state of Monomials.

VERIFY(Valid_State( Constant_Term),0)
VERIFY(Valid_State(   Linear_Term),0)
VERIFY(Valid_State(Quadratic_Term),0)

! Add terms to the Matrix Equation and output.
!
! Given that Phi=2 and Coefficient=3*i  (see above), then...
!
!                           A(i,i)                  B
!
!   General Monomial    C*E*Phi**(E-1)*Vol   C*(E-1)*Phi**E*Vol
```

```
  !
  !   Constant Term                                  -3*i*Vol
  !   Linear Term              3*i*Vol
  !   Quadratic Term          12*i*Vol              12*i*Vol
  !   -----------------------------------------------------------
  !   Sum of Terms            15*i*Vol               9*i*Vol

  call Add_to_Matrix_Equation ( Constant_Term, A_ELLM, B_MV)
  call Add_to_Matrix_Equation (   Linear_Term, A_ELLM, B_MV)
  call Add_to_Matrix_Equation (Quadratic_Term, A_ELLM, B_MV)
  call Output (A_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  call Output (B_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Finalize Monomials, other objects and communications.

  call Finalize (Constant_Term)
  call Finalize (Linear_Term)
  call Finalize (Quadratic_Term)
  call Finalize (Phi_MV)
  call Finalize (Coefficient)
  call Finalize (B_MV)
  call Finalize (X_MV)
  call Finalize (A_ELLM)
  call Finalize (Mesh)
  call Finalize (Comm)

end
```

H.2 Ortho_Diffusion Class Code Listing

The main documentation of the Ortho_Diffusion Class in § 13.2 on page 162 contains additional explanation of this code listing.

```
!
! Author: Michael L. Hall
!         P.O. Box 1663, MS-D413, LANL
!         Los Alamos, NM 87545
!         ph: 505-665-4312
!         email: Hall@LANL.gov
!
! Created on: 03/03/05
! CVS Info:   $Id: ortho_diffusion.F90,v 1.32 2009/09/17 07:26:45 hall Exp $

module Caesar_Ortho_Diffusion_Class

  ! Global use associations.

  use Caesar_Linear_Algebra_Module
  use Caesar_Multi_Mesh_Class
  use Caesar_Numbers_Module, only: zero, half, one, two
  use Caesar_Flags_Module, only: Internal_or_Periodic_BC, Dirichlet_BC, &
                                 Homogeneous_BC, Neumann_BC, Reflective_BC, &
```

```
                                Source_BC, Vacuum_BC, AMR_Large_Cell_BC, &
                                AMR_Small_Cell_BC

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public flags.

  public :: Dirichlet_BC, Homogeneous_BC, Neumann_BC, Reflective_BC, &
            Source_BC, Vacuum_BC

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Add_to_Matrix_Equation, Evaluate, Evaluate_Gradient_Cells, Locus, &
            Name, Output

  interface Initialize
    module procedure Initialize_Ortho_Diffusion
  end interface

  interface Finalize
    module procedure Finalize_Ortho_Diffusion
  end interface

  interface Valid_State
    module procedure Valid_State_Ortho_Diffusion
  end interface

  interface Initialized
    module procedure Initialized_Ortho_Diffusion
  end interface

  interface Add_to_Matrix_Equation
    module procedure Add_to_Matrix_Equation_Ortho_D
  end interface

  interface Evaluate
    module procedure Evaluate_Ortho_Diffusion
  end interface

  interface Evaluate_Gradient_Cells
    module procedure Evaluate_Grad_Cells_Ortho_Diff
  end interface

  interface Locus
    module procedure Get_Locus_Ortho_Diffusion
  end interface

  interface Name
    module procedure Get_Name_Ortho_Diffusion
  end interface
```

```
interface Output
  module procedure Output_Ortho_Diffusion
end interface

! Public type definitions.

public :: Ortho_Diffusion_type

type Ortho_Diffusion_type

  ! Initialization flag.

  type(integer) :: Initialized

  ! The name for this variable.

  type(character,name_length) :: Name

  ! The diffusion coefficient, defined on the cells for now.

  type(real,1) :: Coefficient

  ! The boundary condition flag for each face of each cell.

  type(real,2) :: Boundary_Condition

  ! The boundary condition constant for each face of each cell.

  type(real,2) :: Phi_BC

  ! The extrapolation factor used in the boundary condition calculation.

  type(real) :: Extrapolation

  ! The independent variable at the linearization value (past time
  ! step or iterate).

  type(real,1) :: Phi

  ! Mesh that this ortho_diffusion object is defined on.

  type(Multi_Mesh_type), pointer :: Mesh

  ! Evaluation locus. (to be used in a future version -- for now,
  !                    assumed to be "Cells".)

  type(character,name_length) :: Locus

  ! Locus Base_Structure.

  type(Base_Structure_type), pointer :: Structure

  ! Equation number and total number of equations for this monomial.
```

```
    type(integer) :: Equation, NEquations

  end type Ortho_Diffusion_type

contains
```

The Ortho_Diffusion Class contains the following routines which are listed in separate sections:

Initialize_Ortho_Diffusion (§ H.2.1, page 651)

Finalize_Ortho_Diffusion (§ H.2.2, page 654)

Valid_State_Ortho_Diffusion (§ H.2.3, page 655)

Initialized_Ortho_Diffusion (§ H.2.4, page 656)

Add_to_Matrix_Equation_Ortho_Diffusion (§ H.2.5, page 657)

Evaluate_Ortho_Diffusion (§ H.2.7, page 665)

Evaluate_Gradient_Cells_Ortho_Diffusion (§ H.2.6, page 661)

Get_Harmonic_Diffusion_Coef_Ortho_Diffusion (§ H.2.8, page 669)

Get Value Ortho_Diffusion (§ H.2.9, page 672)

Output_Ortho_Diffusion (§ H.2.10, page 673)

end module Caesar_Ortho_Diffusion_Class

H.2.1 Initialize_Ortho_Diffusion Procedure

The main documentation of the Initialize_Ortho_Diffusion Procedure in § 13.2.1 on page 162 contains additional explanation of this code listing.

```
  subroutine Initialize_Ortho_Diffusion (Diff_Term, Coefficient, &
                                         BC_Faces_of_Cells, &
                                         Phi_BC_Faces_of_Cells, &
                                         Phi_MV, Locus, Mesh, Name, &
                                         Extrapolation, Equation, NEquations, &
                                         status)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variables.

    type(real,1) :: Coefficient                       ! Diffusion coefficient.
    type(character,*), intent(in) :: Locus            ! Evaluation locus.
    type(Multi_Mesh_type), target :: Mesh             ! Diff_Term Mesh.
    ! Old value of the independent variable.
    type(Mathematic_Vector_type), intent(inout) :: Phi_MV
    type(character,*), intent(in), optional :: Name  ! Diff_Term name.
```

```
type(integer,2) :: BC_Faces_of_Cells       ! Boundary condition flags.
type(real,2) :: Phi_BC_Faces_of_Cells      ! Boundary condition constants.
! Factor used in boundary condition calculation.
type(real), intent(in), optional :: Extrapolation
type(integer), intent(in), optional :: Equation   ! Equation number.
type(integer), intent(in), optional :: NEquations ! Number of equations.

! Output variables.

! Diff_Term to be initialized.
type(Ortho_Diffusion_type), intent(out) :: Diff_Term
type(Status_type), intent(out), optional :: status  ! Exit status.

! Internal variables.

type(Status_type), dimension(4) :: allocate_status  ! Allocation Status.
type(Status_type) :: consolidated_status            ! Consolidated Status.
type(integer) :: Faces_per_Cell    ! Number of faces per cell.
type(integer) :: Length_Structure  ! Length of the Structure on this PE.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(.not.Valid_State(Diff_Term),5) ! Diff_Term is not valid.
VERIFY(Valid_State(Coefficient),5)    ! Coefficient is valid.

! Set up pointers.

Diff_Term%Mesh => Mesh
select case (Locus)
case ("Cells")
  Diff_Term%Structure => Cell_Structure(Mesh)
case ("Nodes")
  Diff_Term%Structure => Node_Structure(Mesh)
case ("Faces")
  Diff_Term%Structure => Face_Structure(Mesh)
end select

! Query the mesh.

Faces_per_Cell = Get_Faces_per_Cell(Mesh)
Length_Structure = Length_PE(Diff_Term%Structure)

! Two more requirements: Coefficient and Phi_MV are the right size.

VERIFY(SIZE(Coefficient)==Length_Structure,5)
VERIFY(Length_PE(Phi_MV)==Length_Structure,5)

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)
call Initialize (Diff_Term%Coefficient, Length_Structure, &
```

```
                 allocate_status(1))
call Initialize (Diff_Term%Phi, Length_Structure, allocate_status(2))
call Initialize (Diff_Term%Boundary_Condition, Length_Structure, &
                 Faces_per_Cell, allocate_status(3))
call Initialize (Diff_Term%Phi_BC, Length_Structure, Faces_per_Cell, &
                 allocate_status(4))

! Set up internals.

if (PRESENT(Name)) then
  Diff_Term%Name = Name
else
  Diff_Term%Name = ' '
end if
Diff_Term%Coefficient = Coefficient
Diff_Term%Locus       = Locus
Diff_Term%Phi         = Phi_MV
Diff_Term%Boundary_Condition = BC_Faces_of_Cells
Diff_Term%Phi_BC      = Phi_BC_Faces_of_Cells
if (PRESENT(Extrapolation)) then
  Diff_Term%Extrapolation = Extrapolation
else
  Diff_Term%Extrapolation = half
end if
if (PRESENT(Equation)) then
  Diff_Term%Equation = Equation
else
  Diff_Term%Equation = 1
end if
if (PRESENT(NEquations)) then
  Diff_Term%NEquations = NEquations
else
  Diff_Term%NEquations = 1
end if

! Process status variables.

consolidated_status = allocate_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
  VERIFY(Normal(consolidated_status), 5)
end if
call Finalize (consolidated_status)
call Finalize (allocate_status)

! Set initialization flag.

Diff_Term%Initialized = initialized_flag

! Verify guarantees.

VERIFY(Valid_State(Diff_Term),5)  ! Diff_Term is now valid.
```

```
    return
  end subroutine Initialize_Ortho_Diffusion
```

H.2.2 Finalize_Ortho_Diffusion Procedure

The main documentation of the Finalize_Ortho_Diffusion Procedure in § 13.2.2 on page 163 contains additional explanation of this code listing.

```
  subroutine Finalize_Ortho_Diffusion (Diff_Term, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Diff_Term to be finalized.
    type(Ortho_Diffusion_type), intent(inout) :: Diff_Term

    ! Output variables.

    type(Status_type), intent(out), optional :: status   ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(9) :: deallocate_status ! Deallocation Status.
    type(Status_type) :: consolidated_status             ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Diff_Term),7) ! Diff_Term is valid.

    ! Unset initialization flag.

    Diff_Term%Initialized = uninitialized_flag

    ! Deallocations and finalizations.

    ! Set deallocation status.

    call Initialize (deallocate_status)
    call Initialize (consolidated_status)

    ! Finalize internals.

    NULLIFY(Diff_Term%Mesh)
    NULLIFY(Diff_Term%Structure)
    call Finalize (Diff_Term%Boundary_Condition, deallocate_status(1))
    call Finalize (Diff_Term%Coefficient,        deallocate_status(2))
```

```
    call Finalize (Diff_Term%Locus,              deallocate_status(3))
    call Finalize (Diff_Term%Name,               deallocate_status(4))
    call Finalize (Diff_Term%Extrapolation,      deallocate_status(5))
    call Finalize (Diff_Term%Phi,                deallocate_status(6))
    call Finalize (Diff_Term%Phi_BC,             deallocate_status(7))
    call Finalize (Diff_Term%Equation,           deallocate_status(8))
    call Finalize (Diff_Term%NEquations,         deallocate_status(9))

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(Diff_Term),7) ! Diff_Term is not valid.

    return
  end subroutine Finalize_Ortho_Diffusion
```

H.2.3 Valid_State_Ortho_Diffusion Procedure

The main documentation of the Valid_State_Ortho_Diffusion Procedure in § 13.2.3 on page 164 contains additional explanation of this code listing.

```
  function Valid_State_Ortho_Diffusion (Diff_Term) result(Valid)

    ! Input variables.

    ! Variable to be checked.
    type(Ortho_Diffusion_type), intent(in) :: Diff_Term

    ! Output variables.

    type(logical) :: Valid             ! Logical state.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Start out true.

    Valid = .true.

    ! Check for association of pointered internals.

    Valid = Valid .and. ASSOCIATED(Diff_Term%Mesh)
```

```
    Valid = Valid .and. ASSOCIATED(Diff_Term%Structure)
    if (.not.Valid) return

    ! Check for validity of internals.

    Valid = Valid .and. Initialized(Diff_Term)
    Valid = Valid .and. Valid_State(Diff_Term%Coefficient)
    Valid = Valid .and. Valid_State(Diff_Term%Equation)
    Valid = Valid .and. Valid_State(Diff_Term%Extrapolation)
    Valid = Valid .and. Valid_State(Diff_Term%Locus)
    Valid = Valid .and. Valid_State(Diff_Term%Name)
    Valid = Valid .and. Valid_State(Diff_Term%NEquations)
    Valid = Valid .and. Valid_State(Diff_Term%Phi)
    if (.not.Valid) return

    ! Checks on the validity of Diff_Term.

    Valid = Valid .and. Diff_Term%Extrapolation /= zero
    Valid = Valid .and. &
            (Diff_Term%Equation .InInterval. (/ 1, Diff_Term%NEquations /))

    return
  end function Valid_State_Ortho_Diffusion
```

H.2.4 Initialized_Ortho_Diffusion Procedure

The main documentation of the Initialized_Ortho_Diffusion Procedure in § 13.2.4 on page 164 contains additional explanation of this code listing.

```
  function Initialized_Ortho_Diffusion (Diff_Term) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! Diff_Term to be checked.
    type(Ortho_Diffusion_type), intent(in) :: Diff_Term

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = Diff_Term%Initialized == initialized_flag
```

```
    ! Verify guarantees - none.

    return
  end function Initialized_Ortho_Diffusion
```

H.2.5 Add_to_Matrix_Equation_Ortho_Diffusion Procedure

The main documentation of the Add_to_Matrix_Equation_Ortho_Diffusion Procedure in § 13.2.5 on page 164 contains additional explanation of this code listing.

```
  subroutine Add_to_Matrix_Equation_Ortho_D (Diff_Term, ELLM, RHS_MV)

    ! Input variables.

    ! Diff_Term to be added.
    type(Ortho_Diffusion_type), intent(inout) :: Diff_Term

    ! Input/Output variables.

    ! ELL_Matrix to be incremented.
    type(ELL_Matrix_type), intent(inout) :: ELLM
    ! RHS mathematic vector.
    type(Mathematic_Vector_type), intent(inout) :: RHS_MV

    ! Internal variables.

    ! Arrays to manipulate values of the matrix.
    type(real,2) :: Matrix_Values
    type(real,1) :: RHS_Values
    type(integer,1) :: Matrix_Rows
    type(integer,2) :: Matrix_Columns
    type(integer) :: face, i, other_cell            ! Loop parameters.
    type(integer) :: shift                          ! Index shift.
    type(real,3) :: Area_Faces_of_Cells, Beta
    type(real,2) :: Pseudo_Ortho_Area_F_of_C
    type(real,2) :: Harmonic_Diffusion_Coef_F_of_C  ! Harmonic D at faces.
    type(real,2) :: DeltaR21_Cells_of_Cells, DeltaR1f_Cells_of_Cells
    type(integer) :: Faces_per_Cell  ! Number of faces per cell.
    type(integer) :: NDimensions     ! Number of dimensions.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Diff_Term),5) ! Diff_Term is valid.
    VERIFY(Valid_State(ELLM),5)      ! ELLM is valid.
    VERIFY(Valid_State(RHS_MV),5)    ! RHS_MV is valid.

    ! Query the mesh.

    Faces_per_Cell = Get_Faces_per_Cell (Diff_Term%Mesh)
    NDimensions = Get_NDimensions (Diff_Term%Mesh)
```

```
! Future mods:
!   1. Access routine for Mesh%Cells_of_Cells_Index.
!   2. Other pseudo-ortho methods.

! Create some working vectors.

call Initialize (Matrix_Values, NCells_PE(Diff_Term%Mesh), &
                 1+Faces_per_Cell)
call Initialize (RHS_Values, NCells_PE(Diff_Term%Mesh))
call Initialize (Matrix_Rows, NCells_PE(Diff_Term%Mesh))
call Initialize (Matrix_Columns, NCells_PE(Diff_Term%Mesh), &
                 1+Faces_per_Cell)

! Define Delta R's.

call Initialize (DeltaR21_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Initialize (DeltaR1f_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Get_DeltaR21_Cells_of_Cells (DeltaR21_Cells_of_Cells, Diff_Term%Mesh)
call Get_DeltaR1f_Cells_of_Cells (DeltaR1f_Cells_of_Cells, Diff_Term%Mesh)
!call Get_VecDeltaR21_Cells_of_Cells (DeltaR21_Cells_of_Cells, &
!                                     Diff_Term%Mesh)

! Define Harmonic Average Diffusion Coefficient.

call Initialize (Harmonic_Diffusion_Coef_F_of_C, &
                 NCells_PE(Diff_Term%Mesh), Faces_per_Cell)
call Get_Harmonic_Diffusion_Coef (Harmonic_Diffusion_Coef_F_of_C, &
                                  Diff_Term)

! Must be dimensioned as follows:
! call Initialize (Diff_Term%Boundary_Condition, &
!                  NCells_PE(Diff_Term%Mesh), &
!                  Faces_per_Cell)

! Define Pseudo-Ortho Area. Options are:
!   1. Scalar area.
!   2. Vector area dotted with unit vector joining the cell centers.
!   3. Middle area of face-center box.

call Initialize (Area_Faces_of_Cells, NDimensions, &
                 NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Initialize (Pseudo_Ortho_Area_F_of_C, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Get_Area_Faces_of_Cells (Area_Faces_of_Cells, Diff_Term%Mesh)
if (.true.) then
  ! Option 1, Scalar Area.
  Pseudo_Ortho_Area_F_of_C = ABS(SUM(Area_Faces_of_Cells(:,:,:),1))
end if

! Define Beta values for Boundary Conditions.
```

```
call Initialize (Beta, 3, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
where (Diff_Term%Boundary_Condition(:,:) == Dirichlet_BC)
  Beta(1,:,:) = one
  Beta(2,:,:) = zero
  Beta(3,:,:) = one
elsewhere (Diff_Term%Boundary_Condition(:,:) == Homogeneous_BC)
  Beta(1,:,:) = one
  Beta(2,:,:) = zero
  Beta(3,:,:) = zero
elsewhere (Diff_Term%Boundary_Condition(:,:) == Neumann_BC)
  Beta(1,:,:) = zero
  Beta(2,:,:) = -one
  Beta(3,:,:) = one
elsewhere (Diff_Term%Boundary_Condition(:,:) == Reflective_BC)
  Beta(1,:,:) = zero
  Beta(2,:,:) = -one
  Beta(3,:,:) = zero
elsewhere (Diff_Term%Boundary_Condition(:,:) == Source_BC)
  Beta(1,:,:) = Diff_Term%Extrapolation
  Beta(2,:,:) = one
  Beta(3,:,:) = Diff_Term%Extrapolation
elsewhere (Diff_Term%Boundary_Condition(:,:) == Vacuum_BC)
  Beta(1,:,:) = Diff_Term%Extrapolation
  Beta(2,:,:) = one
  Beta(3,:,:) = zero
end where

! Set Matrix_Rows, Matrix_Columns.
!
! To spread out the rows and columns from this equation into a multiple
! equation matrix, translate local numbers via:
!   i --> NEquations*(i - 1) + Equation
! and global numbers via:
!   i + shift --> NEquations*(i + shift - 1) + Equation

Matrix_Columns(:,2:) = Diff_Term%Mesh%Cells_of_Cells_Index
Matrix_Columns(:,2:) = Diff_Term%NEquations * (Matrix_Columns(:,2:) - 1) &
                       + Diff_Term%Equation
shift = First_PE(Diff_Term%Structure) - 1
do i = 1, Length_PE(Diff_Term%Structure)
  Matrix_Rows(i) = Diff_Term%NEquations*(i - 1) + Diff_Term%Equation
  Matrix_Columns(i,1) = Diff_Term%NEquations*(i + shift - 1) + &
                        Diff_Term%Equation
end do

! Calculate matrix coefficients for F . A .
!                                    f   f
! Matrix structure:
!   Matrix_Values(:,1) = Diagonal, the cell center value.
!   Matrix_Values(:,2:Faces_per_Cell+1) = The values for the cells on
!     the other side of each face, shifted by one from the face number
!     to allow for the diagonal entry.
```

```
Matrix_Values = zero
do face = 1, Faces_per_Cell
  other_cell = face  ! Another name for clearer semantics.

  where (Diff_Term%Boundary_Condition(:,face) == Internal_or_Periodic_BC)

    ! For the internal faces and periodic boundary conditions:
    !
    !                     ( Phi  - Phi  )
    !                h    (    1      2 )
    !   F . A  =  A  D     ---------------
    !    f   f     f  12    | delta r   |
    !                       |        12 |
    !
    ! The other_cell (Phi_2) coefficient is calculated and stored
    ! here. The cell (diagonal entry, Phi_1) coefficient is calculated
    ! below by summing and subtracting all the other_cell coefficients.

    Matrix_Values(:,other_cell+1) = &
      - Pseudo_Ortho_Area_F_of_C(:,face) &
      * Harmonic_Diffusion_Coef_F_of_C(:,face) &
      / DeltaR21_Cells_of_Cells(:,face)

  elsewhere

    ! For the generalized Robin Boundary Conditions:
    !
    !               A  ( Beta  Phi  - Beta  Phi   )
    !                f (     1    1       3    BC )
    !   F . A   =  ------------------------------------
    !    f   f      ( Beta  + Beta  | delta r   | / D  )
    !               (     2      1 |       1f |    1 )
    !
    ! The cell (diagonal entry, Phi_1) coefficient and the constant, RHS,
    ! value are both calculated here. Note the sign flip on the RHS value
    ! to move it to the right-hand side.
    !
    ! Since a Fick's law extrapolation is used to determine this
    ! term, if D_1 is set to zero the flow (F.A) is set to zero,
    ! regardless of the Beta values.

    where (Diff_Term%Coefficient(:) /= zero)
      Matrix_Values(:,1) = Matrix_Values(:,1) + &
        Pseudo_Ortho_Area_F_of_C(:,face) &
        * Beta(1,:,face) &
        / (Beta(2,:,face) &
        + Beta(1,:,face) * DeltaR1f_Cells_of_Cells(:,face) &
        / Diff_Term%Coefficient(:) )

      RHS_Values(:) = RHS_Values(:) + &
        Pseudo_Ortho_Area_F_of_C(:,face) &
        * Beta(3,:,face) * Diff_Term%Phi_BC(:,face) &
        / (Beta(2,:,face) &
```

```
          + Beta(1,:,face) * DeltaR1f_Cells_of_Cells(:,face) &
          / Diff_Term%Coefficient(:) )
      end where

    end where
  end do

  ! Finish the cell (diagonal entry, Phi_1) coefficient calculation by
  ! by summing and subtracting all the other_cell coefficients.

  Matrix_Values(:,1) = Matrix_Values(:,1) - SUM(Matrix_Values(:,2:),2)

  ! Add the values to the ELLM matrix and RHS vector.

  call Add_Values (ELLM, Matrix_Values, Matrix_Rows, Matrix_Columns, &
                   Global=.false.)
  call Add_Values (RHS_MV, RHS_Values, Matrix_Rows, Global=.false.)

  ! Finalize working structures.

  call Finalize (DeltaR21_Cells_of_Cells)
  call Finalize (DeltaR1f_Cells_of_Cells)
  call Finalize (Harmonic_Diffusion_Coef_F_of_C)
  call Finalize (Pseudo_Ortho_Area_F_of_C)
  call Finalize (Beta)
  call Finalize (Area_Faces_of_Cells)
  call Finalize (Matrix_Values)
  call Finalize (RHS_Values)
  call Finalize (Matrix_Rows)
  call Finalize (Matrix_Columns)

  ! Verify guarantees.

  VERIFY(Valid_State(ELLM),5)     ! ELLM is valid.
  VERIFY(Valid_State(RHS_MV),5)   ! RHS_MV is valid.

  return
end subroutine Add_to_Matrix_Equation_Ortho_D
```

H.2.6 Evaluate_Gradient_Cells_Ortho_Diffusion Procedure

The main documentation of the Evaluate_Gradient_Cells_Ortho_Diffusion Procedure in § 13.2.6 on page 165 contains additional explanation of this code listing.

```
subroutine Evaluate_Grad_Cells_Ortho_Diff (Diff_Term, Phi_MV, Grad_Cells)

  ! Input variables.

  type(Ortho_Diffusion_type), intent(inout) :: Diff_Term ! Diffusion term.
  type(Mathematic_Vector_type), intent(inout) :: Phi_MV  ! Phi math vector.

  ! Output variable.
```

```
! Vector Gradients at each cell center.
type(real,2) :: Grad_Cells

! Internal variables.

type(integer) :: dim, face, other_cell       ! Loop parameters.
! Vector Gradients dotted with the normals at each face of each cell.
type(real,2) :: Grad_dot_N_Faces_of_Cells
type(real,3) :: Beta
type(real,2) :: Harmonic_Diffusion_Coef_F_of_C
type(real,2) :: DeltaR21_Cells_of_Cells, DeltaR1f_Cells_of_Cells
type(integer) :: Faces_per_Cell  ! Number of faces per cell.
type(integer) :: NDimensions     ! Number of dimensions.

! Phi manipulation variables.
type(real,1) :: Phi_Cells
type(Distributed_Vector_type) :: Phi_Cells_DV
type(Collected_Array_type) :: Phi_Cells_of_Cells_CA
type(real,2) :: Phi_Cells_of_Cells

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Diff_Term),5)           ! Diff_Term is valid.
VERIFY(Valid_State(Phi_MV),5)              ! Phi_MV is valid.
VERIFY(Valid_State(Grad_Cells),5)          ! Grad_Cells is valid.
! Grad_Cells is dimensioned correctly.
VERIFY(SIZE(Grad_Cells,1) == Get_NDimensions(Diff_Term%Mesh),5)
VERIFY(SIZE(Grad_Cells,2) == NCells_PE(Diff_Term%Mesh),5)

! Query the mesh.

Faces_per_Cell = Get_Faces_per_Cell (Diff_Term%Mesh)
NDimensions = Get_NDimensions (Diff_Term%Mesh)

! Future mods:
!   1. Access routine for Mesh%Cells_of_Cells_Index.

! Define Delta R's.

call Initialize (DeltaR21_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Initialize (DeltaR1f_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Get_DeltaR21_Cells_of_Cells (DeltaR21_Cells_of_Cells, Diff_Term%Mesh)
call Get_DeltaR1f_Cells_of_Cells (DeltaR1f_Cells_of_Cells, Diff_Term%Mesh)

! Define Harmonic Average Diffusion Coefficient.

call Initialize (Harmonic_Diffusion_Coef_F_of_C, &
                 NCells_PE(Diff_Term%Mesh), Faces_per_Cell)
call Get_Harmonic_Diffusion_Coef (Harmonic_Diffusion_Coef_F_of_C, &
```

```
                                  Diff_Term)

    ! Get Phi for Cells and Cells of Cells.

    call Initialize (Phi_Cells, NCells_PE(Diff_Term%Mesh))
    call Initialize (Phi_Cells_DV, Diff_Term%Structure, 1)
    call Initialize (Phi_Cells_of_Cells_CA, &
                     Diff_Term%Mesh%Cells_of_Cells_Index, 1)
    call Initialize (Phi_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                     Faces_per_Cell)
    Phi_Cells = Phi_MV
    Phi_Cells_DV = Phi_Cells
    Phi_Cells_of_Cells_CA = Phi_Cells_DV
    Phi_Cells_of_Cells = Phi_Cells_of_Cells_CA
    call Finalize (Phi_Cells_DV)
    call Finalize (Phi_Cells_of_Cells_CA)

    ! Define Beta values for Boundary Conditions.

    call Initialize (Beta, 3, NCells_PE(Diff_Term%Mesh), &
                     Faces_per_Cell)
    where (Diff_Term%Boundary_Condition(:,:) == Dirichlet_BC)
      Beta(1,:,:) = one
      Beta(2,:,:) = zero
      Beta(3,:,:) = one
    elsewhere (Diff_Term%Boundary_Condition(:,:) == Homogeneous_BC)
      Beta(1,:,:) = one
      Beta(2,:,:) = zero
      Beta(3,:,:) = zero
    elsewhere (Diff_Term%Boundary_Condition(:,:) == Neumann_BC)
      Beta(1,:,:) = zero
      Beta(2,:,:) = -one
      Beta(3,:,:) = one
    elsewhere (Diff_Term%Boundary_Condition(:,:) == Reflective_BC)
      Beta(1,:,:) = zero
      Beta(2,:,:) = -one
      Beta(3,:,:) = zero
    elsewhere (Diff_Term%Boundary_Condition(:,:) == Source_BC)
      Beta(1,:,:) = Diff_Term%Extrapolation
      Beta(2,:,:) = one
      Beta(3,:,:) = Diff_Term%Extrapolation
    elsewhere (Diff_Term%Boundary_Condition(:,:) == Vacuum_BC)
      Beta(1,:,:) = Diff_Term%Extrapolation
      Beta(2,:,:) = one
      Beta(3,:,:) = zero
    end where

    ! Calculate Grad Phi . N.
    !
    !                                1
    ! Grad Phi . N (outward) =  -  --- F . N
    !         f   f                D   f   f
    !                               1
```

```
call Initialize (Grad_dot_N_Faces_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
Grad_dot_N_Faces_of_Cells = zero
do face = 1, Faces_per_Cell
  other_cell = face  ! Another name for clearer semantics.

  where (Diff_Term%Coefficient(:) == zero)

    Grad_dot_N_Faces_of_Cells(:,face) = zero

  elsewhere (Diff_Term%Boundary_Condition(:,face) == &
             Internal_or_Periodic_BC)

    ! For the internal faces and periodic boundary conditions:
    !
    !                                h
    !                               D     ( Phi  - Phi  )
    !                                12   (    1      2 )
    !  Grad Phi . N (outward) =  -  -----  ---------------
    !          f   f                 D     | delta r   |
    !                                 1    |       12 |

    Grad_dot_N_Faces_of_Cells(:,face) = &
      - Harmonic_Diffusion_Coef_F_of_C(:,face) &
      / Diff_Term%Coefficient(:) &
      * (Phi_Cells(:) - Phi_Cells_of_Cells(:,other_cell)) &
      / DeltaR21_Cells_of_Cells(:,face)

  elsewhere

    ! For the generalized Robin Boundary Conditions:
    !
    !                                  ( Beta  Phi  - Beta  Phi   )
    !                                  (     1    1       3    BC )
    ! Grad Phi . N (outward) =  -  ---------------------------------------
    !         f   f                D  ( Beta  + Beta  | delta r   | / D  )
    !                               1 (     2       1 |       1f |    1 )

    Grad_dot_N_Faces_of_Cells(:,face) = &
      - (Beta(1,:,face) * Phi_Cells(:) &
      - Beta(3,:,face) * Diff_Term%Phi_BC(:,face) ) &
      / Diff_Term%Coefficient(:) &
      / (Beta(2,:,face) &
      + Beta(1,:,face) * DeltaR1f_Cells_of_Cells(:,face) &
      / Diff_Term%Coefficient(:) )

  end where
end do

! Multiply by N (outward unit normal, simply a plus or minus sign) and
! average face values to get cell center value for each direction.

do dim = 1, NDimensions
  Grad_Cells(dim,:) = (- Grad_dot_N_Faces_of_Cells(:,2*dim-1) &
```

```
                           + Grad_dot_N_Faces_of_Cells(:,2*dim  )) &
                          / two
    end do
!write(6,*) 'Phi_Cells(5)                        = ', Phi_Cells(5)
!write(6,*) 'Phi_Cells_of_Cells(5,1)             = ', Phi_Cells_of_Cells(5,1)
!write(6,*) 'Phi_Cells_of_Cells(5,2)             = ', Phi_Cells_of_Cells(5,2)
!write(6,*) 'Grad_dot_N_Faces_of_Cells(5,1)      = ', Grad_dot_N_Faces_of_Cells(5,1)
!write(6,*) 'Grad_dot_N_Faces_of_Cells(5,2)      = ', Grad_dot_N_Faces_of_Cells(5,2)
!write(6,*) 'Grad_Cells(1,5) = ', Grad_Cells(1,5)
!write(6,*) 'DeltaR21_Cells_of_Cells(5,1)        = ', DeltaR21_Cells_of_Cells(5,1)
!write(6,*) 'DeltaR21_Cells_of_Cells(5,2)        = ', DeltaR21_Cells_of_Cells(5,2)
!write(6,*) 'Diff_Term%Coefficient(5)            = ', Diff_Term%Coefficient(5)
!write(6,*) 'Harmonic_Diffusion_Coef_F_of_C(5,2) = ', Harmonic_Diffusion_Coef_F_of_C(5,2)


    ! Finalize working structures.

    call Finalize (DeltaR21_Cells_of_Cells)
    call Finalize (DeltaR1f_Cells_of_Cells)
    call Finalize (Harmonic_Diffusion_Coef_F_of_C)
    call Finalize (Beta)
    call Finalize (Phi_Cells)
    call Finalize (Phi_Cells_of_Cells)
    call Finalize (Grad_dot_N_Faces_of_Cells)

    ! Verify guarantees.

    VERIFY(Valid_State(Grad_Cells),5)     ! Grad_Cells is valid.
    VERIFY(Valid_State(Phi_MV),5)         ! Phi_MV is valid.
    VERIFY(Valid_State(Diff_Term),5)      ! Diff_Term is valid.

    return
  end subroutine Evaluate_Grad_Cells_Ortho_Diff
```

H.2.7 Evaluate_Ortho_Diffusion Procedure

The main documentation of the Evaluate_Ortho_Diffusion Procedure in § 13.2.7 on page 166 contains additional explanation of this code listing.

```
  subroutine Evaluate_Ortho_Diffusion (Diff_Term, Phi_MV, Div_D_Grad_Phi_Cells)

    ! Input variables.

    type(Ortho_Diffusion_type), intent(inout) :: Diff_Term ! Diffusion term.
    type(Mathematic_Vector_type), intent(inout) :: Phi_MV  ! Phi math vector.

    ! Output variable.

    type(real,1) :: Div_D_Grad_Phi_Cells  ! Div D Grad at each cell center.

    ! Internal variables.
```

```
type(integer) :: face, other_cell              ! Loop parameters.
type(real,3) :: Area_Faces_of_Cells, Beta
type(real,2) :: Pseudo_Ortho_Area_F_of_C
type(real,2) :: Harmonic_Diffusion_Coef_F_of_C  ! Harmonic D at faces.
type(real,2) :: DeltaR21_Cells_of_Cells, DeltaR1f_Cells_of_Cells
type(integer) :: Faces_per_Cell                ! Number of faces per cell.
type(integer) :: NDimensions                   ! Number of dimensions.
type(real,1) :: Volume_Cells                   ! Volume of the cells.

! Phi manipulation variables.
type(real,1) :: Phi_Cells
type(Distributed_Vector_type) :: Phi_Cells_DV
type(Collected_Array_type) :: Phi_Cells_of_Cells_CA
type(real,2) :: Phi_Cells_of_Cells

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Diff_Term),5)           ! Diff_Term is valid.
VERIFY(Valid_State(Phi_MV),5)              ! Phi_MV is valid.
VERIFY(Valid_State(Div_D_Grad_Phi_Cells),5) ! Div_D_Grad_Phi_Cells is valid.
! Div_D_Grad_Phi_Cells is dimensioned correctly.
VERIFY(SIZE(Div_D_Grad_Phi_Cells,1) == NCells_PE(Diff_Term%Mesh),5)

! Query the mesh.

Faces_per_Cell = Get_Faces_per_Cell (Diff_Term%Mesh)
NDimensions = Get_NDimensions (Diff_Term%Mesh)

! Future mods:
!   1. Access routine for Mesh%Cells_of_Cells_Index.

! Define Delta R's.

call Initialize (DeltaR21_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Initialize (DeltaR1f_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Get_DeltaR21_Cells_of_Cells (DeltaR21_Cells_of_Cells, Diff_Term%Mesh)
call Get_DeltaR1f_Cells_of_Cells (DeltaR1f_Cells_of_Cells, Diff_Term%Mesh)

! Define Harmonic Average Diffusion Coefficient.

call Initialize (Harmonic_Diffusion_Coef_F_of_C, &
                 NCells_PE(Diff_Term%Mesh), Faces_per_Cell)
call Get_Harmonic_Diffusion_Coef (Harmonic_Diffusion_Coef_F_of_C, &
                                  Diff_Term)

! Define Pseudo-Ortho Area. Options are:
!   1. Scalar area.
!   2. Vector area dotted with unit vector joining the cell centers.
!   3. Middle area of face-center box.
```

```
call Initialize (Area_Faces_of_Cells, NDimensions, &
                 NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Initialize (Pseudo_Ortho_Area_F_of_C, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Get_Area_Faces_of_Cells (Area_Faces_of_Cells, Diff_Term%Mesh)
if (.true.) then
  ! Option 1, Scalar Area.
  Pseudo_Ortho_Area_F_of_C = ABS(SUM(Area_Faces_of_Cells(:,:,:),1))
end if

! Get cell volumes.

call Initialize (Volume_Cells, NCells_PE(Diff_Term%Mesh))
call Get_Volume_Cells (Volume_Cells, Diff_Term%Mesh)

! Get Phi for Cells and Cells of Cells.

call Initialize (Phi_Cells, NCells_PE(Diff_Term%Mesh))
call Initialize (Phi_Cells_DV, Diff_Term%Structure, 1)
call Initialize (Phi_Cells_of_Cells_CA, &
                 Diff_Term%Mesh%Cells_of_Cells_Index, 1)
call Initialize (Phi_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
Phi_Cells = Phi_MV
Phi_Cells_DV = Phi_Cells
Phi_Cells_of_Cells_CA = Phi_Cells_DV
Phi_Cells_of_Cells = Phi_Cells_of_Cells_CA
call Finalize (Phi_Cells_DV)
call Finalize (Phi_Cells_of_Cells_CA)

! Define Beta values for Boundary Conditions.

call Initialize (Beta, 3, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
where (Diff_Term%Boundary_Condition(:,:) == Dirichlet_BC)
  Beta(1,:,:) = one
  Beta(2,:,:) = zero
  Beta(3,:,:) = one
elsewhere (Diff_Term%Boundary_Condition(:,:) == Homogeneous_BC)
  Beta(1,:,:) = one
  Beta(2,:,:) = zero
  Beta(3,:,:) = zero
elsewhere (Diff_Term%Boundary_Condition(:,:) == Neumann_BC)
  Beta(1,:,:) = zero
  Beta(2,:,:) = -one
  Beta(3,:,:) = one
elsewhere (Diff_Term%Boundary_Condition(:,:) == Reflective_BC)
  Beta(1,:,:) = zero
  Beta(2,:,:) = -one
  Beta(3,:,:) = zero
elsewhere (Diff_Term%Boundary_Condition(:,:) == Source_BC)
  Beta(1,:,:) = Diff_Term%Extrapolation
  Beta(2,:,:) = one
```

```
    Beta(3,:,:) = Diff_Term%Extrapolation
  elsewhere (Diff_Term%Boundary_Condition(:,:) == Vacuum_BC)
    Beta(1,:,:) = Diff_Term%Extrapolation
    Beta(2,:,:) = one
    Beta(3,:,:) = zero
  end where

  ! Calculate -Div D Grad Phi, by integrating over the cell and
  ! dividing by the volume:
  !
  !                            ---
  !                        1   \
  !   - Div D Grad Phi  =  ---  /    F . A
  !                  c     V   ---   f   f
  !                         c faces
  !

  Div_D_Grad_Phi_Cells = zero
  do face = 1, Faces_per_Cell
    other_cell = face  ! Another name for clearer semantics.

    where (Diff_Term%Coefficient(:) == zero)

      ! No op. Flux is zero.

    elsewhere (Diff_Term%Boundary_Condition(:,face) == &
               Internal_or_Periodic_BC)

      ! For the internal faces and periodic boundary conditions:
      !
      !                   h
      !   F . A      A  D     ( Phi  - Phi  )
      !    f   f      f  12   (    1      2 )
      !   ------  =  ------   ---------------
      !      V         V       | delta r   |
      !       c         c      |       12 |

      Div_D_Grad_Phi_Cells(:) = Div_D_Grad_Phi_Cells(:) + &
        Pseudo_Ortho_Area_F_of_C(:,face) / Volume_Cells(:) &
        * Harmonic_Diffusion_Coef_F_of_C(:,face) &
        * (Phi_Cells(:) - Phi_Cells_of_Cells(:,other_cell)) &
        / DeltaR21_Cells_of_Cells(:,face)

    elsewhere

      ! For the generalized Robin Boundary Conditions:
      !
      !   F . A        A  ( Beta  Phi  - Beta  Phi   )
      !    f   f        f (     1    1       3    BC )
      !   ------  =  ----------------------------------------
      !      V        V  ( Beta  + Beta  | delta r   | / D  )
      !       c        c (     2       1 |       1f |    1 )

      Div_D_Grad_Phi_Cells(:) = Div_D_Grad_Phi_Cells(:) + &
```

```
        Pseudo_Ortho_Area_F_of_C(:,face) / Volume_Cells(:) &
        * (Beta(1,:,face) * Phi_Cells(:) &
        - Beta(3,:,face) * Diff_Term%Phi_BC(:,face) ) &
        / (Beta(2,:,face) &
        + Beta(1,:,face) * DeltaR1f_Cells_of_Cells(:,face) &
        / Diff_Term%Coefficient(:) )

    end where
  end do

  ! Finalize working structures.

  call Finalize (Area_Faces_of_Cells)
  call Finalize (Pseudo_Ortho_Area_F_of_C)
  call Finalize (Volume_Cells)
  call Finalize (DeltaR21_Cells_of_Cells)
  call Finalize (DeltaR1f_Cells_of_Cells)
  call Finalize (Harmonic_Diffusion_Coef_F_of_C)
  call Finalize (Beta)
  call Finalize (Phi_Cells)
  call Finalize (Phi_Cells_of_Cells)

  ! Verify guarantees.

  VERIFY(Valid_State(Div_D_Grad_Phi_Cells),5) ! Div_D_Grad_Phi_Cells is valid.
  VERIFY(Valid_State(Phi_MV),5)               ! Phi_MV is valid.
  VERIFY(Valid_State(Diff_Term),5)            ! Diff_Term is valid.

  return
end subroutine Evaluate_Ortho_Diffusion
```

H.2.8 Get_Harmonic_Diffusion_Coef_Ortho_Diffusion Procedure

The main documentation of the Get_Harmonic_Diffusion_Coef_Ortho_Diffusion Procedure in § 13.2.8 on page 166 contains additional explanation of this code listing.

```
subroutine Get_Harmonic_Diffusion_Coef (Harmonic_Diffusion_Coef_F_of_C, &
                                        Diff_Term)

  ! Input variable.

  type(Ortho_Diffusion_type), intent(inout) :: Diff_Term ! Diffusion term.

  ! Output variable.

  ! Harmonic diffusion coefficient at each face of each cell.
  type(real,2) :: Harmonic_Diffusion_Coef_F_of_C

  ! Internal variables.

  type(integer) :: face, other_cell      ! Loop parameters.
  ! Diffusion coefficient variables for cells and cells of cells.
```

```
type(Collected_Array_type) :: Coefficient_Cells_of_Cells_CA
type(Distributed_Vector_type) :: Coefficient_Cells_DV
type(real,2) :: Coefficient_Cells_of_Cells
! Absolute distances between the cell centers and the faces.
type(real,2) :: DeltaR21_Cells_of_Cells, DeltaR1f_Cells_of_Cells, &
                DeltaR2f_Cells_of_Cells
! A toggle for different harmonic averages.
type(character,name_length) :: harmonic_diffusion_option
type(integer) :: Faces_per_Cell  ! Number of faces per cell.
type(integer) :: NDimensions     ! Number of dimensions.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Diff_Term),5)            ! Diff_Term is valid.
! Harmonic_Diffusion_Coef_F_of_C is valid.
VERIFY(Valid_State(Harmonic_Diffusion_Coef_F_of_C),5)
! Harmonic_Diffusion_Coef_F_of_C is dimensioned correctly.
VERIFY(SIZE(Harmonic_Diffusion_Coef_F_of_C,1) == dnl
       NCells_PE(Diff_Term%Mesh),5)
VERIFY(SIZE(Harmonic_Diffusion_Coef_F_of_C,2) == dnl
       Get_Faces_per_Cell(Diff_Term%Mesh),5)

! Query the mesh.

Faces_per_Cell = Get_Faces_per_Cell (Diff_Term%Mesh)
NDimensions = Get_NDimensions (Diff_Term%Mesh)

! Future mods:
!   1. Access routine for Mesh%Cells_of_Cells_Index.
!   2. Other pseudo-ortho methods.

! Define Delta R's.

call Initialize (DeltaR21_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Initialize (DeltaR1f_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Initialize (DeltaR2f_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
call Get_DeltaR21_Cells_of_Cells (DeltaR21_Cells_of_Cells, Diff_Term%Mesh)
call Get_DeltaR1f_Cells_of_Cells (DeltaR1f_Cells_of_Cells, Diff_Term%Mesh)
call Get_DeltaR2f_Cells_of_Cells (DeltaR2f_Cells_of_Cells, Diff_Term%Mesh)

! Get the Diffusion Coefficient for the neighboring cells.

call Initialize (Coefficient_Cells_DV, Diff_Term%Structure, 1)
call Initialize (Coefficient_Cells_of_Cells_CA, &
                 Diff_Term%Mesh%Cells_of_Cells_Index, 1)
call Initialize (Coefficient_Cells_of_Cells, NCells_PE(Diff_Term%Mesh), &
                 Faces_per_Cell)
Coefficient_Cells_DV = Diff_Term%Coefficient
Coefficient_Cells_of_Cells_CA = Coefficient_Cells_DV
```

```
    Coefficient_Cells_of_Cells = Coefficient_Cells_of_Cells_CA
    call Finalize (Coefficient_Cells_of_Cells_CA)
    call Finalize (Coefficient_Cells_DV)

    ! Define Harmonic Average Diffusion Coefficient. Four options:
    !
    !   alpha - DeltaR21 (DeltaR1f/D1 + DeltaR2F/D2)^-1
    !   beta  - (DeltaR1f + DeltaR2F) (DeltaR1f/D1 + DeltaR2F/D2)^-1
    !   gamma - (V1 + V2) (V2/D1 + V1/D2)^-1
    !   delta - (V1 + V2) (V1/D1 + V2/D2)^-1
    !
    ! Options alpha, beta, and delta reduce to the orthogonal method on
    ! orthogonal meshes. Options beta, gamma, and delta reduce to the
    ! constant D value if D is constant, regardless of the geometry. Option
    ! gamma reproduces a bug in Telluride, and is here for comparison
    ! purposes only (don't use it).

    harmonic_diffusion_option = 'beta'   ! Hardwired to beta for now.
    do face = 1, Faces_per_Cell
      other_cell = face  ! Another name for clearer semantics.

      if (harmonic_diffusion_option == 'alpha') then
        where (Diff_Term%Coefficient(:) == zero .OR. &
               Coefficient_Cells_of_Cells(:,other_cell) == zero)
          Harmonic_Diffusion_Coef_F_of_C(:,face) = zero
        elsewhere
          Harmonic_Diffusion_Coef_F_of_C(:,face) = &
            DeltaR21_Cells_of_Cells(:,face) * &
            (DeltaR1f_Cells_of_Cells(:,face) / &
             Diff_Term%Coefficient(:) + &
             DeltaR2f_Cells_of_Cells(:,face) / &
             Coefficient_Cells_of_Cells(:,other_cell))**(-1)
        end where
      else if (harmonic_diffusion_option == 'beta') then
        where (Diff_Term%Coefficient(:) == zero .OR. &
               Coefficient_Cells_of_Cells(:,other_cell) == zero)
          Harmonic_Diffusion_Coef_F_of_C(:,face) = zero
        elsewhere
          Harmonic_Diffusion_Coef_F_of_C(:,face) = &
            (DeltaR1f_Cells_of_Cells(:,face) + &
             DeltaR2f_Cells_of_Cells(:,face)) * &
            (DeltaR1f_Cells_of_Cells(:,face) / &
             Diff_Term%Coefficient(:) + &
             DeltaR2f_Cells_of_Cells(:,face) / &
             Coefficient_Cells_of_Cells(:,other_cell))**(-1)
        end where
      else
        VERIFY(.false.,1) ! Not implemented yet.
      end if
    end do

    ! Finalize working structures.

    call Finalize (DeltaR21_Cells_of_Cells)
```

```
    call Finalize (DeltaR1f_Cells_of_Cells)
    call Finalize (DeltaR2f_Cells_of_Cells)
    call Finalize (Coefficient_Cells_of_Cells)

    ! Verify guarantees.

    ! Harmonic_Diffusion_Coef_F_of_C is valid.
    VERIFY(Valid_State(Harmonic_Diffusion_Coef_F_of_C),5)
    VERIFY(Valid_State(Diff_Term),5)      ! Diff_Term is valid.

    return
  end subroutine Get_Harmonic_Diffusion_Coef
```

H.2.9 Get Value Ortho_Diffusion Functions

The main documentation of the Get Value Ortho_Diffusion Functions in § 13.2.9 on page 166 contains additional explanation of this code listing.

```
  define([CHARACTER_ACCESS_ROUTINE],[
    pushdef([VALUE], [$1])
    pushdef([VALUE_Result], expand(VALUE_Result))
    pushdef([Get_CHARACTER_VALUE_Ortho_Diffusion], dnl
             expand(Get_VALUE_Ortho_Diffusion))

    function Get_CHARACTER_VALUE_Ortho_Diffusion (Diff_Term) &
                                               result(VALUE_Result)

      ! Input variables.

      type(Ortho_Diffusion_type), intent(in) :: Diff_Term   ! Diff_Term object.

      ! Output variables.

      type(character,80) :: VALUE_Result        ! Diff_Term value to be output.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(Diff_Term),5)               ! Diff_Term is valid.

      ! Set value.

      VALUE_Result = Diff_Term%Value

      ! Verify guarantees - none.

      return
    end function Get_CHARACTER_VALUE_Ortho_Diffusion

    popdef([VALUE])
    popdef([VALUE_Result])
```

```
    popdef([Get_CHARACTER_VALUE_Ortho_Diffusion])
  ])

  fortext([Value],
          [Locus Name],[
    CHARACTER_ACCESS_ROUTINE(Value)
  ])
```

H.2.10 Output_Ortho_Diffusion Procedure

The main documentation of the Output_Ortho_Diffusion Procedure in § 13.2.10 on page 167 contains additional explanation of this code listing.

```
  subroutine Output_Ortho_Diffusion (Diff_Term, First, Last, Unit, Indent, &
                                     status)

    ! Input variables.

    ! Variable to be output.
    type(Ortho_Diffusion_type), intent(inout) :: Diff_Term
    type(integer), intent(in), optional :: First    ! Extents of value data
    type(integer), intent(in), optional :: Last     !   to be output.
    type(integer), intent(in), optional :: Unit     ! Output unit.
    type(integer), optional :: Indent               ! Indentation.

    ! Output variable.

    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(integer) :: A_First                         ! Actual first value.
    type(integer) :: A_Last                          ! Actual last value.
    type(integer) :: A_Unit                          ! Actual output unit.
    type(integer) :: A_Indent                        ! Actual indentation.
    type(character,80) :: Blanks                     ! A line of blanks.
    type(Mathematic_Vector_type) :: Coefficient_MV   ! Temp MV for Coefficient.
    type(Mathematic_Vector_type) :: Phi_MV           ! Temp MV for Phi.
    type(Status_type), dimension(2) :: allocate_status ! Allocation Status.
    type(Status_type) :: consolidated_status           ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Diff_Term),5)                  ! Diff_Term is valid.

    ! Set unit number.

    if (PRESENT(Unit)) then
      A_Unit = Unit
    else
```

```
    A_Unit = 6
  end if

  ! Set indentation.

  if (PRESENT(Indent)) then
    A_Indent = Indent
  else
    A_Indent = 0
  end if
  Blanks = ' '

  ! Set up local limits in terms of global limits.

  if (PRESENT(First)) then
    A_First = First
  else
    A_First = 1
  end if
  if (PRESENT(Last)) then
    A_Last = Last
  else
    A_Last = Length_Total(Diff_Term%Structure)
  end if

  ! Allocations and initializations.

  call Initialize (allocate_status)
  call Initialize (consolidated_status)

  ! Output Identification Info.

  if (this_is_IO_PE) then
    write (A_Unit,100) Blanks(1:A_Indent), 'Ortho_Diffusion Information:'
    write (A_Unit,101) Blanks(1:A_Indent+2), 'Name                = ', &
                       TRIM(Diff_Term%Name)
    write (A_Unit,102) Blanks(1:A_Indent+2), 'Initialized         =', &
                       Initialized(Diff_Term)
    write (A_Unit,101) Blanks(1:A_Indent+2), 'Locus               = ', &
                       TRIM(Diff_Term%Locus)
    write (A_Unit,103) Blanks(1:A_Indent+2), 'Extrapolation       = ', &
                       Diff_Term%Extrapolation
    write (A_Unit,104) Blanks(1:A_Indent+2), 'Equation            =', &
                       Diff_Term%Equation
    write (A_Unit,104) Blanks(1:A_Indent+2), 'NEquations          =', &
                       Diff_Term%NEquations
  end if

  ! Output internal structure info.

  call Output (Diff_Term%Structure, A_Unit, 'Ortho_Diffusion Locus', &
               A_Indent+2)

  call Initialize (Coefficient_MV, Diff_Term%Structure, &
```

```
                   'Coefficient as an MV', allocate_status(1))
    Coefficient_MV = Diff_Term%Coefficient
    call Output (Coefficient_MV, A_First, A_Last, A_Unit, A_Indent+2)
    call Finalize (Coefficient_MV)

    call Initialize (Phi_MV, Diff_Term%Structure, &
                   'Phi as an MV', allocate_status(2))
    Phi_MV = Diff_Term%Phi
    call Output (Phi_MV, A_First, A_Last, A_Unit, A_Indent+2)
    call Finalize (Phi_MV)

    ! Process status variables.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Format statements.

    100 format (/, 2a, /)
    101 format (3a)
    102 format (2a, l2)
    103 format (2a, 1p, e13.5e3)
    104 format (2a, 1p, i4)

    ! Verify guarantees - none.

    return
  end subroutine Output_Ortho_Diffusion
```

H.2.11 Ortho_Diffusion Class Unit Test Program

This lightly commented program performs a unit test on the Ortho_Diffusion Class, which is described in § 13.2 on page 162.

```
program Unit_Test

  use Caesar_Linear_Algebra_Module
  use Caesar_Multi_Mesh_Class
  use Caesar_Ortho_Diffusion_Class
  use Caesar_Monomial_Class
  use Caesar_Numbers_Module, only: zero, half, one, two, three, four, &
                                   six, eight, twelve, sixteen
  implicit none

  type(Communication_type) :: Comm
```

```
type(Status_type) :: status
type(integer) :: c, NRows

! Mesh variables.

type(real,2) :: Coordinates_Cells          ! Coordinates of the cell centers.
type(real,1) :: Lengths                    ! Physical extent of the
                                           ! domain in each direction.
type(Multi_Mesh_type) :: Mesh              ! Mesh object.
type(integer) :: NDimensions               ! Number of dimensions.
type(integer) :: NCells_X                  ! Number of cells in X.
type(integer) :: NCells_Y                  ! Number of cells in Y.
type(integer) :: NCells_Z                  ! Number of cells in Z.
type(integer) :: Faces_per_Cell            ! Number of local faces per cell.
type(integer,2) :: Flag_Faces_of_Cells     ! Mesh boundary flags.
type(real,3) :: Coordinates_Faces_of_Cells ! Coordinates of face centers.
type(real) :: X                            ! Local X coordinate.

! Physics variables.

type(real,1) :: Diffusion_Coefficient       ! Diffusion Coefficient.
type(Mathematic_Vector_type) :: Phi_MV      ! Diffused Quantity (MV).
type(real,1) :: Phi                         ! Diffused Quantity (BNV).
type(real,2) :: Gradient_Cells              ! Gradient vectors at the cells.
type(real) :: Phi_0, Phi_1                  ! Dirichlet BC constants.
type(real,1) :: Sigma_a                     ! Absorption cross-section.
type(real,1) :: Source                      ! Source term.
type(integer,2) :: BC_Faces_of_Cells        ! Boundary condition flags.
type(real,2) :: Phi_BC_Faces_of_Cells       ! Boundary condition constants.
type(real) :: Extrapolation                 ! Boundary condition factor.
type(real) :: D0, D1, D2, Q0

! Equation variables.

type(Ortho_Diffusion_type) :: Ortho_Diff_Term      ! Diffusion term.
type(Monomial_type) :: Linear_Term                 ! Absorption term.
type(Monomial_type) :: Constant_Term               ! Source term.
type(logical) :: Derivative                        ! Time derivative toggle.
type(real) :: Delta_t                              ! Time step size.
type(real) :: Exponent             ! Exponent used in setting Monomial terms.

! Matrix variables.

type(ELL_Matrix_type) :: Matrix_ELLM  ! Diffusion equation matrix.
type(integer) :: MaxNonzeros    ! Maximum number of nonzero entries per row.
type(real,1) :: X_BNV                       ! Solution (BNV).
type(Mathematic_Vector_type) :: X_MV        ! Solution (MV).
type(Mathematic_Vector_type) :: RHS_MV ! Right-hand side of diffusion eqn.
type(Mathematic_Vector_type) :: Error_MV    ! Error Vector (MV).
type(Mathematic_Vector_type) :: Exact_MV    ! Exact Vector (MV).
type(real,1) :: Exact                       ! Exact Vector (BNV).
type(real) :: Relative_L2_Error_Norm        ! Relative L2 Error Norm.
! Base structures for the matrix.
type(Base_Structure_type) :: Row_Structure, Column_Structure
```

```
type(Solver_type) :: Solver
type(integer) :: Variable                  ! Variable number.
type(integer) :: Equation, eq              ! Equation number.
type(integer) :: NEquations                ! Number of equations.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Initializations.

call Initialize (Comm)
call Output (Comm)
call Initialize (status)

! Create a mesh.

NCells_X = 10
NCells_Y = 2
NCells_Z = 12
!NCells_X = 2
!NCells_Y = 1
!NCells_Z = 1
NDimensions = 3
call Initialize (Lengths, NDimensions)
Lengths = (/ 1.d0, 1.d0, 1.d0 /)

call Initialize (Mesh, NDimensions, Lengths, NCells_X, NCells_Y, NCells_Z, &
                 'Uniform Mesh', status)
call Initialize (Coordinates_Cells, NDimensions, NCells_PE(Mesh), &
                 status)
call Get_Coordinates_Cells (Coordinates_Cells, Mesh)
Delta_t = zero
Derivative = .false.
Variable = 3
Equation = 2
NEquations = 3

! Set Diffusion Coefficient for every face of each cell.
!  Faces_of_Cells way -- for later:
!
!Faces_per_Cell = two*NDimensions
!call Initialize (Diffusion_Coefficient, NCells_PE(Mesh), Faces_per_Cell, &
!                 status(4))
!do lf = 1, Faces_per_Cell
!  where (Coordinates_Cells(1,:) < 0.5)
!    Diffusion_Coefficient(:,lf) = Diff_Coeff(1)
!  elsewhere
!    Diffusion_Coefficient(:,lf) = Diff_Coeff(2)
!  end where
!end do

! Set Boundary Conditions.
! (Should do by face, but doing by faces_of_cells is easier for now.)
! call Initialize (Boundary_Conditions, NFaces_PE(Mesh), status)
```

```
Faces_per_Cell = Get_Faces_per_Cell (Mesh)
call Initialize (BC_Faces_of_Cells, NCells_PE(Mesh), Faces_per_Cell, &
                 status)
call Initialize (Flag_Faces_of_Cells, NCells_PE(Mesh), Faces_per_Cell, &
                 status)
call Initialize (Phi_BC_Faces_of_Cells, NCells_PE(Mesh), Faces_per_Cell, &
                 status)
call Get_Flag_Faces_of_Cells (Flag_Faces_of_Cells, Mesh)
! Extrapolation Factor is set to the default value -- cannot leave it out of
! the initialize call because it appears before the status variable.
Extrapolation = half

! Get Cell Face Coordinates.

call Initialize (Coordinates_Faces_of_Cells, NDimensions, &
                 NCells_PE(Mesh), Faces_per_Cell)
call Get_Coordinates_Faces_of_Cells (Coordinates_Faces_of_Cells, Mesh)

! Physics initializations.

call Initialize (Sigma_a, NCells_PE(Mesh), status)
call Initialize (Source, NCells_PE(Mesh), status)
call Initialize (Diffusion_Coefficient, NCells_PE(Mesh), status)
call Initialize (Phi, NCells_PE(Mesh), status)
call Initialize (Gradient_Cells, NDimensions, NCells_PE(Mesh), status)
call Initialize (Exact, NCells_PE(Mesh), status)
call Initialize (Phi_MV, Cell_Structure(Mesh), 'Phi_Cells', status)
call Initialize (Error_MV, Cell_Structure(Mesh), 'Error = Solution - Exact', &
                 status)
call Initialize (Exact_MV, Cell_Structure(Mesh), 'Exact', status)

! Set up Matrix Equation.

MaxNonzeros = 2*NDimensions + 1
call Generate_Multiple (Row_Structure, NEquations, Cell_Structure(Mesh), &
                        'Rows')
call Generate_Multiple (Column_Structure, NEquations, Cell_Structure(Mesh), &
                        'Columns')
call Initialize (Matrix_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 'Coefficients', status)
call Initialize (X_BNV, Length_PE(Column_Structure), status)
call Initialize (X_MV, Column_Structure, 'Solution', status)
call Initialize (RHS_MV, Row_Structure, 'Right-Hand Side', &
                 status)

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
! Problem X.1: 1-D Linear Solution with Dirichlet BCs
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (this_is_IO_PE) then
  write (6,100) 'Problem X.1: 1-D Linear Solution with Dirichlet BCs'
end if

! Problem parameters.
```

```
Phi_0 = 47.d0
D0 = one/3.d0
Q0 = zero

! Set Absorption (Removal) cross-section for each cell.

Sigma_a = zero

! Set Source term for each cell.

Source = -Q0

! Set Diffusion Coefficient for each cell.

Diffusion_Coefficient = D0

! Set Boundary Conditions.

where (Flag_Faces_of_Cells == 1)     ! Left (-x) Face.
  BC_Faces_of_Cells = Dirichlet_BC
  Phi_BC_Faces_of_Cells = Phi_0
elsewhere (Flag_Faces_of_Cells == 2) ! Right (+x) Face.
  BC_Faces_of_Cells = Homogeneous_BC
elsewhere (Flag_Faces_of_Cells == 3) ! Front (-y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 4) ! Back (+y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 5) ! Bottom (-z) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 6) ! Top (+z) Face.
  BC_Faces_of_Cells = Reflective_BC
end where

! Set Phi and X to an initial guess to avoid division by zero error in
! LAMG in the stopping test.

Phi = one
Phi_MV = Phi
X_BNV = one
X_MV = X_BNV

! Define Steady State Diffusion Equation.

call Initialize (Ortho_Diff_Term, Diffusion_Coefficient, BC_Faces_of_Cells, &
                 Phi_BC_Faces_of_Cells, Phi_MV, 'Cells', Mesh, &
                 'Diffusion Term', Extrapolation, Equation, NEquations, &
                 status)
Exponent = one
call Initialize (Linear_Term, Sigma_a, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Absorption Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)
Exponent = zero
call Initialize (Constant_Term, Source, Exponent, Phi_MV, 'Cells', Mesh, &
```

```
                 'Source Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)

  ! Output Terms.

  NRows = NCells_PE(Mesh)
  !call Output ( Constant_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  !call Output (   Linear_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  !call Output (Ortho_Diff_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Check state of Equation Terms.

  VERIFY(Valid_State(  Constant_Term),0)
  VERIFY(Valid_State(    Linear_Term),0)
  VERIFY(Valid_State(Ortho_Diff_Term),0)

  ! Populate Matrix Equation.

  call Add_to_Matrix_Equation (Ortho_Diff_Term, Matrix_ELLM, RHS_MV)
  call Add_to_Matrix_Equation (Linear_Term, Matrix_ELLM, RHS_MV)
  call Add_to_Matrix_Equation (Constant_Term, Matrix_ELLM, RHS_MV)
  call Set_0_Diagonal_to_1 (Matrix_ELLM)

  ! Solve Matrix Equation.

  VERIFY(Valid_State(Matrix_ELLM),0)
  !call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  call Initialize (Solver, 'LAMG')
  call Set (Solver, 'Epsilon', 1.d-10)
  call Solve (Solver, Matrix_ELLM, X_MV, RHS_MV)
  call Finalize (Solver)

  ! Output.

  VERIFY(Valid_State(X_MV),0)
  !call Output (X_MV)
  !call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  VERIFY(Valid_State(RHS_MV),0)
  !call Output (RHS_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Check answers.

  X_BNV = X_MV
  Phi = X_BNV(Equation::NEquations)
  Phi_MV = Phi
  do c = 1, NCells_PE(Mesh)
    Exact(c) = Phi_0*(one-Coordinates_Cells(1,c))
  ! write (6,*) c, Coordinates_Cells(1,c), Phi(c), Exact(c), Exact(c) - Phi(c)
  end do
  Error_MV = Phi - Exact
  call Output (Error_MV, 0, 0)
  Exact_MV = Exact
  Relative_L2_Error_Norm = Norm(Error_MV) / Norm(Exact_MV)
  if (this_is_IO_PE) &
```

```
    write (6,102) '  Relative L2 error norm = ', Relative_L2_Error_Norm

  ! Plotting output (turn on if desired).

  if (.false.) then
    call Dump_GMV ('GMVout.gmv', Mesh, Phi_MV, Exact_MV, Error_MV)
  end if

  ! Check gradient.

  call Evaluate_Gradient_Cells (Ortho_Diff_Term, Phi_MV, Gradient_Cells)
  do c = 1, NCells_PE(Mesh)
    if (ABS(Gradient_Cells(1,c) - (-Phi_0)) > 9.d-8 .or. &
        ABS(Gradient_Cells(2,c) - (zero)  ) > 2.d-8 .or. &
        ABS(Gradient_Cells(3,c) - (zero)  ) > 5.d-8) then
      write (6,*) 'Error (Problem X.1): Grad = ', Gradient_Cells(:,c)
    end if
  end do

  ! Finalize terms.

  call Finalize (Ortho_Diff_Term)
  call Finalize (Linear_Term)
  call Finalize (Constant_Term)

  ! Reset entire Matrix Equation.
  !
  ! Note that we must finalize X_MV before Matrix_ELLM because the call to
  ! Solver may use a MatVec to calculate a residual, which would create an
  ! OV in X_MV based on the Data_Index in Matrix_ELLM. Tricky.

  call Finalize (X_MV)
  call Finalize (RHS_MV)
  call Finalize (Matrix_ELLM)
  call Initialize (Matrix_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                   'Coefficients', status)
  call Initialize (RHS_MV, Row_Structure, 'Right-Hand Side', &
                   status)
  call Initialize (X_MV, Column_Structure, 'Solution', status)

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  ! Problem X.2: 1-D Linear Solution with Source/Vacuum BCs
  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  if (this_is_IO_PE) then
    write (6,100) 'Problem X.2: 1-D Linear Solution with Source/Vacuum BCs'
  end if

  ! Problem parameters.

  Phi_0 = 47.d0
  D0 = one/3.d0
  Q0 = zero
```

```
! Set Absorption (Removal) cross-section for each cell.

Sigma_a = zero

! Set Source term for each cell.

Source = -Q0

! Set Diffusion Coefficient for each cell.

Diffusion_Coefficient = D0

! Set Boundary Conditions.

where (Flag_Faces_of_Cells == 1)     ! Left (-x) Face.
  BC_Faces_of_Cells = Source_BC
  Phi_BC_Faces_of_Cells = Phi_0
elsewhere (Flag_Faces_of_Cells == 2) ! Right (+x) Face.
  BC_Faces_of_Cells = Vacuum_BC
elsewhere (Flag_Faces_of_Cells == 3) ! Front (-y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 4) ! Back (+y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 5) ! Bottom (-z) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 6) ! Top (+z) Face.
  BC_Faces_of_Cells = Reflective_BC
end where

! Set Phi and X to an initial guess to avoid division by zero error in
! LAMG in the stopping test.

Phi = one
Phi_MV = Phi
X_BNV = one
X_MV = X_BNV

! Define Steady State Diffusion Equation.

call Initialize (Ortho_Diff_Term, Diffusion_Coefficient, BC_Faces_of_Cells, &
                 Phi_BC_Faces_of_Cells, Phi_MV, 'Cells', Mesh, &
                 'Diffusion Term', Extrapolation, Equation, NEquations, &
                 status)
Exponent = one
call Initialize (Linear_Term, Sigma_a, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Absorption Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)
Exponent = zero
call Initialize (Constant_Term, Source, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Source Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)

! Output Terms.
```

```
NRows = NCells_PE(Mesh)
!call Output ( Constant_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (   Linear_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (Ortho_Diff_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check state of Equation Terms.

VERIFY(Valid_State(  Constant_Term),0)
VERIFY(Valid_State(    Linear_Term),0)
VERIFY(Valid_State(Ortho_Diff_Term),0)

! Populate Matrix Equation.

call Add_to_Matrix_Equation (Ortho_Diff_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Linear_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Constant_Term, Matrix_ELLM, RHS_MV)
call Set_0_Diagonal_to_1 (Matrix_ELLM)

! Solve Matrix Equation.

VERIFY(Valid_State(Matrix_ELLM),0)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Initialize (Solver, 'LAMG')
call Set (Solver, 'Epsilon', 1.d-10)
call Solve (Solver, Matrix_ELLM, X_MV, RHS_MV)
call Finalize (Solver)

! Output.

VERIFY(Valid_State(X_MV),0)
!call Output (X_MV)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
VERIFY(Valid_State(RHS_MV),0)
!call Output (RHS_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check answers.

X_BNV = X_MV
Phi = X_BNV(Equation::NEquations)
Phi_MV = Phi
do c = 1, NCells_PE(Mesh)
  Exact(c) = Phi_0 * (one + two*D0 - Coordinates_Cells(1,c)) / &
             (one + four*D0)
! write (6,*) c, Coordinates_Cells(1,c), Phi(c), Exact(c), Exact(c) - Phi(c)
end do
Error_MV = Phi - Exact
call Output (Error_MV, 0, 0)
Exact_MV = Exact
Relative_L2_Error_Norm = Norm(Error_MV) / Norm(Exact_MV)
if (this_is_IO_PE) &
  write (6,102) '  Relative L2 error norm = ', Relative_L2_Error_Norm

! Plotting output (turn on if desired).
```

```
if (.false.) then
  call Dump_GMV ('GMVout.gmv', Mesh, Phi_MV, Exact_MV, Error_MV)
end if

! Check gradient.

call Evaluate_Gradient_Cells (Ortho_Diff_Term, Phi_MV, Gradient_Cells)
do c = 1, NCells_PE(Mesh)
  if (ABS(Gradient_Cells(1,c) - (-Phi_0/(one + four*D0))) > 5.d-9 .or. &
      ABS(Gradient_Cells(2,c) - (zero)  ) > 4.d-9 .or. &
      ABS(Gradient_Cells(3,c) - (zero)  ) > 4.d-9) then
    write (6,*) 'Error (Problem X.2): Grad = ', Gradient_Cells(:,c)
  end if
end do

! Finalize terms.

call Finalize (Ortho_Diff_Term)
call Finalize (Linear_Term)
call Finalize (Constant_Term)

! Reset entire Matrix Equation.
!
! Note that we must finalize X_MV before Matrix_ELLM because the call to
! Solver may use a MatVec to calculate a residual, which would create an
! OV in X_MV based on the Data_Index in Matrix_ELLM. Tricky.

call Finalize (X_MV)
call Finalize (RHS_MV)
call Finalize (Matrix_ELLM)
call Initialize (Matrix_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 'Coefficients', status)
call Initialize (RHS_MV, Row_Structure, 'Right-Hand Side', &
                 status)
call Initialize (X_MV, Column_Structure, 'Solution', status)

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
! Problem X.3: Multi-D Linear Solution with Dirichlet BCs
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (this_is_IO_PE) then
  write (6,100) 'Problem X.3: Multi-D Linear Solution with Dirichlet BCs'
end if

! Problem parameters.

Phi_0 = 47.d0
D0 = one/3.d0
Q0 = zero

! Set Absorption (Removal) cross-section for each cell.

Sigma_a = zero
```

```
! Set Source term for each cell.

Source = -Q0

! Set Diffusion Coefficient for each cell.

Diffusion_Coefficient = D0

! Set Boundary Conditions.

where (Flag_Faces_of_Cells == 1)     ! Left (-x) Face.
  BC_Faces_of_Cells = Dirichlet_BC
elsewhere (Flag_Faces_of_Cells == 2) ! Right (+x) Face.
  BC_Faces_of_Cells = Dirichlet_BC
elsewhere (Flag_Faces_of_Cells == 3) ! Front (-y) Face.
  BC_Faces_of_Cells = Dirichlet_BC
elsewhere (Flag_Faces_of_Cells == 4) ! Back (+y) Face.
  BC_Faces_of_Cells = Dirichlet_BC
elsewhere (Flag_Faces_of_Cells == 5) ! Bottom (-z) Face.
  BC_Faces_of_Cells = Dirichlet_BC
elsewhere (Flag_Faces_of_Cells == 6) ! Top (+z) Face.
  BC_Faces_of_Cells = Dirichlet_BC
end where
Phi_BC_Faces_of_Cells(:,:) = Phi_0 * SUM(Coordinates_Faces_of_Cells(:,:,:),1)

! Set Phi and X to an initial guess to avoid division by zero error in
! LAMG in the stopping test.

Phi = one
Phi_MV = Phi
X_BNV = one
X_MV = X_BNV

! Define Steady State Diffusion Equation.

call Initialize (Ortho_Diff_Term, Diffusion_Coefficient, BC_Faces_of_Cells, &
                 Phi_BC_Faces_of_Cells, Phi_MV, 'Cells', Mesh, &
                 'Diffusion Term', Extrapolation, Equation, NEquations, &
                 status)
Exponent = one
call Initialize (Linear_Term, Sigma_a, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Absorption Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)
Exponent = zero
call Initialize (Constant_Term, Source, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Source Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)

! Output Terms.

NRows = NCells_PE(Mesh)
!call Output ( Constant_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (   Linear_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (Ortho_Diff_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
```

```
! Check state of Equation Terms.

VERIFY(Valid_State(  Constant_Term),0)
VERIFY(Valid_State(    Linear_Term),0)
VERIFY(Valid_State(Ortho_Diff_Term),0)

! Populate Matrix Equation.

call Add_to_Matrix_Equation (Ortho_Diff_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Linear_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Constant_Term, Matrix_ELLM, RHS_MV)
call Set_0_Diagonal_to_1 (Matrix_ELLM)

! Solve Matrix Equation.

VERIFY(Valid_State(Matrix_ELLM),0)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Initialize (Solver, 'LAMG')
call Set (Solver, 'Epsilon', 1.d-10)
call Solve (Solver, Matrix_ELLM, X_MV, RHS_MV)
call Finalize (Solver)

! Output.

VERIFY(Valid_State(X_MV),0)
!call Output (X_MV)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
VERIFY(Valid_State(RHS_MV),0)
!call Output (RHS_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check answers.

X_BNV = X_MV
Phi = X_BNV(Equation::NEquations)
Phi_MV = Phi
do c = 1, NCells_PE(Mesh)
  Exact(c) = Phi_0 * SUM(Coordinates_Cells(:,c),1)
! write (6,*) c, Coordinates_Cells(1,c), Phi(c), Exact(c), Exact(c) - Phi(c)
end do
Error_MV = Phi - Exact
call Output (Error_MV, 0, 0)
Exact_MV = Exact
Relative_L2_Error_Norm = Norm(Error_MV) / Norm(Exact_MV)
if (this_is_IO_PE) &
  write (6,102) '  Relative L2 error norm = ', Relative_L2_Error_Norm

! Plotting output (turn on if desired).

if (.false.) then
  call Dump_GMV ('GMVout.gmv', Mesh, Phi_MV, Exact_MV, Error_MV)
end if

! Check gradient.
```

```
call Evaluate_Gradient_Cells (Ortho_Diff_Term, Phi_MV, Gradient_Cells)
do c = 1, NCells_PE(Mesh)
  if (ABS(Gradient_Cells(1,c) - (Phi_0) ) > 7.d-7 .or. &
      ABS(Gradient_Cells(2,c) - (Phi_0) ) > 7.d-7 .or. &
      ABS(Gradient_Cells(3,c) - (Phi_0) ) > 7.d-7) then
    write (6,*) 'Error (Problem X.3): Grad = ', Gradient_Cells(:,c)
  end if
end do

! Finalize terms.

call Finalize (Ortho_Diff_Term)
call Finalize (Linear_Term)
call Finalize (Constant_Term)

! Reset entire Matrix Equation.
!
! Note that we must finalize X_MV before Matrix_ELLM because the call to
! Solver may use a MatVec to calculate a residual, which would create an
! OV in X_MV based on the Data_Index in Matrix_ELLM. Tricky.

call Finalize (X_MV)
call Finalize (RHS_MV)
call Finalize (Matrix_ELLM)
call Initialize (Matrix_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 'Coefficients', status)
call Initialize (RHS_MV, Row_Structure, 'Right-Hand Side', &
                 status)
call Initialize (X_MV, Column_Structure, 'Solution', status)

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
! Problem X.4: 1-D Quadratic Solution with Dirichlet BCs
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (this_is_IO_PE) then
  write (6,100) 'Problem X.4: 1-D Quadratic Solution with Dirichlet BCs'
end if

! Problem parameters.

D0 = one/3.d0
Q0 = 23.d0
Phi_0 = 6.d0
Phi_1 = 19.d0

! Set Absorption (Removal) cross-section for each cell.

Sigma_a = zero

! Set Source term for each cell.

Source = -Q0
```

```
! Set Diffusion Coefficient for each cell.

Diffusion_Coefficient = D0

! Set Boundary Conditions.

Phi_BC_Faces_of_Cells = zero
where (Flag_Faces_of_Cells == 1)     ! Left (-x) Face.
  BC_Faces_of_Cells = Dirichlet_BC
  Phi_BC_Faces_of_Cells = Phi_0
elsewhere (Flag_Faces_of_Cells == 2) ! Right (+x) Face.
  BC_Faces_of_Cells = Dirichlet_BC
  Phi_BC_Faces_of_Cells = Phi_1
elsewhere (Flag_Faces_of_Cells == 3) ! Front (-y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 4) ! Back (+y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 5) ! Bottom (-z) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 6) ! Top (+z) Face.
  BC_Faces_of_Cells = Reflective_BC
end where

! Set Phi and X to an initial guess to avoid division by zero error in
! LAMG in the stopping test.

Phi = one
Phi_MV = Phi
X_BNV = one
X_MV = X_BNV

! Define Steady State Diffusion Equation.

call Initialize (Ortho_Diff_Term, Diffusion_Coefficient, BC_Faces_of_Cells, &
                 Phi_BC_Faces_of_Cells, Phi_MV, 'Cells', Mesh, &
                 'Diffusion Term', Extrapolation, Equation, NEquations, &
                 status)
Exponent = one
call Initialize (Linear_Term, Sigma_a, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Absorption Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)
Exponent = zero
call Initialize (Constant_Term, Source, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Source Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)

! Output Terms.

NRows = NCells_PE(Mesh)
!call Output ( Constant_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (   Linear_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (Ortho_Diff_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check state of Equation Terms.
```

```
VERIFY(Valid_State(  Constant_Term),0)
VERIFY(Valid_State(    Linear_Term),0)
VERIFY(Valid_State(Ortho_Diff_Term),0)

! Populate Matrix Equation.

call Add_to_Matrix_Equation (Ortho_Diff_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Linear_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Constant_Term, Matrix_ELLM, RHS_MV)
call Set_0_Diagonal_to_1 (Matrix_ELLM)

! Solve Matrix Equation.

VERIFY(Valid_State(Matrix_ELLM),0)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Initialize (Solver, 'LAMG')
call Set (Solver, 'Epsilon', 1.d-10)
call Solve (Solver, Matrix_ELLM, X_MV, RHS_MV)
call Finalize (Solver)

! Output.

VERIFY(Valid_State(X_MV),0)
!call Output (X_MV)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
VERIFY(Valid_State(RHS_MV),0)
!call Output (RHS_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check answers.
! Note: I tested problem X.4 and it converged by 2nd Order when
! refined in the X-direction.

X_BNV = X_MV
Phi = X_BNV(Equation::NEquations)
Phi_MV = Phi
do c = 1, NCells_PE(Mesh)
  X = Coordinates_Cells(1,c)
  Exact(c) = Q0 / (two*D0) * (X - X**2) + (Phi_1 - Phi_0) * X + Phi_0
  !write (6,*) c, Coordinates_Cells(1,c), Phi(c), Exact(c), Exact(c) - Phi(c)
end do
Error_MV = Phi - Exact
call Output (Error_MV, 0, 0)
Exact_MV = Exact
Relative_L2_Error_Norm = Norm(Error_MV) / Norm(Exact_MV)
if (this_is_IO_PE) &
  write (6,102) '  Relative L2 error norm = ', Relative_L2_Error_Norm

! Plotting output (turn on if desired).

if (.false.) then
  call Dump_GMV ('GMVout.gmv', Mesh, Phi_MV, Exact_MV, Error_MV)
end if
```

```
! Check gradient.

call Evaluate_Gradient_Cells (Ortho_Diff_Term, Phi_MV, Gradient_Cells)
do c = 1, NCells_PE(Mesh)
  X = Coordinates_Cells(1,c)
  if (ABS(Gradient_Cells(1,c) - &
      (Q0 / (two*D0) * (1 - 2*X) + (Phi_1 - Phi_0)) ) > 1.d-8 .or. &
      ABS(Gradient_Cells(2,c) - (zero) ) > 1.d-8 .or. &
      ABS(Gradient_Cells(3,c) - (zero) ) > 1.d-8) then
    write (6,*) 'Error (Problem X.4): Grad = ', Gradient_Cells(:,c)
  end if
end do

! Finalize terms.

call Finalize (Ortho_Diff_Term)
call Finalize (Linear_Term)
call Finalize (Constant_Term)

! Reset entire Matrix Equation.
!
! Note that we must finalize X_MV before Matrix_ELLM because the call to
! Solver may use a MatVec to calculate a residual, which would create an
! OV in X_MV based on the Data_Index in Matrix_ELLM. Tricky.

call Finalize (X_MV)
call Finalize (RHS_MV)
call Finalize (Matrix_ELLM)
call Initialize (Matrix_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 'Coefficients', status)
call Initialize (RHS_MV, Row_Structure, 'Right-Hand Side', &
                 status)
call Initialize (X_MV, Column_Structure, 'Solution', status)

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
! Problem X.5: 1-D Quadratic Solution with Vacuum BCs
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (this_is_IO_PE) then
  write (6,100) 'Problem X.5: 1-D Quadratic Solution with Vacuum BCs'
end if

! Problem parameters.

D0 = one/3.d0
Q0 = 23.d0

! Set Absorption (Removal) cross-section for each cell.

Sigma_a = zero

! Set Source term for each cell.

Source = -Q0
```

```
! Set Diffusion Coefficient for each cell.

Diffusion_Coefficient = D0

! Set Boundary Conditions.

where (Flag_Faces_of_Cells == 1)     ! Left (-x) Face.
  BC_Faces_of_Cells = Vacuum_BC
elsewhere (Flag_Faces_of_Cells == 2) ! Right (+x) Face.
  BC_Faces_of_Cells = Vacuum_BC
elsewhere (Flag_Faces_of_Cells == 3) ! Front (-y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 4) ! Back (+y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 5) ! Bottom (-z) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 6) ! Top (+z) Face.
  BC_Faces_of_Cells = Reflective_BC
end where
Phi_BC_Faces_of_Cells = zero

! Set Phi and X to an initial guess to avoid division by zero error in
! LAMG in the stopping test.

Phi = one
Phi_MV = Phi
X_BNV = one
X_MV = X_BNV

! Define Steady State Diffusion Equation.

call Initialize (Ortho_Diff_Term, Diffusion_Coefficient, BC_Faces_of_Cells, &
                 Phi_BC_Faces_of_Cells, Phi_MV, 'Cells', Mesh, &
                 'Diffusion Term', Extrapolation, Equation, NEquations, &
                 status)
Exponent = one
call Initialize (Linear_Term, Sigma_a, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Absorption Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)
Exponent = zero
call Initialize (Constant_Term, Source, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Source Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)

! Output Terms.

NRows = NCells_PE(Mesh)
!call Output ( Constant_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (   Linear_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (Ortho_Diff_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check state of Equation Terms.
```

```
VERIFY(Valid_State(  Constant_Term),0)
VERIFY(Valid_State(    Linear_Term),0)
VERIFY(Valid_State(Ortho_Diff_Term),0)

! Populate Matrix Equation.

call Add_to_Matrix_Equation (Ortho_Diff_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Linear_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Constant_Term, Matrix_ELLM, RHS_MV)
call Set_0_Diagonal_to_1 (Matrix_ELLM)

! Solve Matrix Equation.

VERIFY(Valid_State(Matrix_ELLM),0)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Initialize (Solver, 'LAMG')
call Set (Solver, 'Epsilon', 1.d-10)
call Solve (Solver, Matrix_ELLM, X_MV, RHS_MV)
call Finalize (Solver)

! Output.

VERIFY(Valid_State(X_MV),0)
!call Output (X_MV)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
VERIFY(Valid_State(RHS_MV),0)
!call Output (RHS_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check answers.
! Note: I tested problem X.5 and it converged by 2nd Order when
! refined in the X-direction.

X_BNV = X_MV
Phi = X_BNV(Equation::NEquations)
Phi_MV = Phi
do c = 1, NCells_PE(Mesh)
  X = Coordinates_Cells(1,c)
  Exact(c) = Q0 / (two*D0) * ( two*D0 + X - X**2)
! write (6,*) c, Coordinates_Cells(1,c), Phi(c), Exact(c), Exact(c) - Phi(c)
end do
Error_MV = Phi - Exact
call Output (Error_MV, 0, 0)
Exact_MV = Exact
Relative_L2_Error_Norm = Norm(Error_MV) / Norm(Exact_MV)
if (this_is_IO_PE) &
  write (6,102) '  Relative L2 error norm = ', Relative_L2_Error_Norm

! Plotting output (turn on if desired).

if (.false.) then
  call Dump_GMV ('GMVout.gmv', Mesh, Phi_MV, Exact_MV, Error_MV)
end if

! Check gradient.
```

```
call Evaluate_Gradient_Cells (Ortho_Diff_Term, Phi_MV, Gradient_Cells)
do c = 1, NCells_PE(Mesh)
  X = Coordinates_Cells(1,c)
  if (ABS(Gradient_Cells(1,c) - (Q0 / (two*D0) * (1 - 2*X)) ) > 1.d-8 .or. &
      ABS(Gradient_Cells(2,c) - (zero) ) > 1.d-8 .or. &
      ABS(Gradient_Cells(3,c) - (zero) ) > 1.d-8) then
    write (6,*) 'Error (Problem X.5): Grad = ', Gradient_Cells(:,c)
  end if
end do

! Finalize terms.

call Finalize (Ortho_Diff_Term)
call Finalize (Linear_Term)
call Finalize (Constant_Term)

! Reset entire Matrix Equation.
!
! Note that we must finalize X_MV before Matrix_ELLM because the call to
! Solver may use a MatVec to calculate a residual, which would create an
! OV in X_MV based on the Data_Index in Matrix_ELLM. Tricky.

call Finalize (X_MV)
call Finalize (RHS_MV)
call Finalize (Matrix_ELLM)
call Initialize (Matrix_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 'Coefficients', status)
call Initialize (RHS_MV, Row_Structure, 'Right-Hand Side', &
                 status)
call Initialize (X_MV, Column_Structure, 'Solution', status)

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
! Problem X.6: 1-D Quartic Solution with Vacuum BCs
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (this_is_IO_PE) then
  write (6,100) 'Problem X.6: 1-D Quartic Solution with Vacuum BCs'
end if

! Problem parameters.

D0 = one/3.d0
Q0 = 23.d0

! Set Absorption (Removal) cross-section for each cell.

Sigma_a = zero

! Set Source term for each cell.

do c = 1, NCells_PE(Mesh)
  Source(c) = -Q0 * Coordinates_Cells(1,c)**2
end do
```

```
! Set Diffusion Coefficient for each cell.

Diffusion_Coefficient = D0

! Set Boundary Conditions.

where (Flag_Faces_of_Cells == 1)     ! Left (-x) Face.
  BC_Faces_of_Cells = Vacuum_BC
elsewhere (Flag_Faces_of_Cells == 2) ! Right (+x) Face.
  BC_Faces_of_Cells = Vacuum_BC
elsewhere (Flag_Faces_of_Cells == 3) ! Front (-y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 4) ! Back (+y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 5) ! Bottom (-z) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 6) ! Top (+z) Face.
  BC_Faces_of_Cells = Reflective_BC
end where
Phi_BC_Faces_of_Cells = zero

! Set Phi and X to an initial guess to avoid division by zero error in
! LAMG in the stopping test.

Phi = one
Phi_MV = Phi
X_BNV = one
X_MV = X_BNV

! Define Steady State Diffusion Equation.

call Initialize (Ortho_Diff_Term, Diffusion_Coefficient, BC_Faces_of_Cells, &
                 Phi_BC_Faces_of_Cells, Phi_MV, 'Cells', Mesh, &
                 'Diffusion Term', Extrapolation, Equation, NEquations, &
                 status)
Exponent = one
call Initialize (Linear_Term, Sigma_a, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Absorption Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)
Exponent = zero
call Initialize (Constant_Term, Source, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Source Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)

! Output Terms.

NRows = NCells_PE(Mesh)
!call Output ( Constant_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (   Linear_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
!call Output (Ortho_Diff_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check state of Equation Terms.
```

```
VERIFY(Valid_State(  Constant_Term),0)
VERIFY(Valid_State(    Linear_Term),0)
VERIFY(Valid_State(Ortho_Diff_Term),0)

! Populate Matrix Equation.

call Add_to_Matrix_Equation (Ortho_Diff_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Linear_Term, Matrix_ELLM, RHS_MV)
call Add_to_Matrix_Equation (Constant_Term, Matrix_ELLM, RHS_MV)
call Set_0_Diagonal_to_1 (Matrix_ELLM)

! Solve Matrix Equation.

VERIFY(Valid_State(Matrix_ELLM),0)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
call Initialize (Solver, 'LAMG')
call Set (Solver, 'Epsilon', 1.d-10)
call Solve (Solver, Matrix_ELLM, X_MV, RHS_MV)
call Finalize (Solver)

! Output.

VERIFY(Valid_State(X_MV),0)
!call Output (X_MV)
!call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
VERIFY(Valid_State(RHS_MV),0)
!call Output (RHS_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

! Check answers.
! Note: I tested problem X.6 and it converged by 2nd Order when
! refined in the X-direction.

X_BNV = X_MV
Phi = X_BNV(Equation::NEquations)
Phi_MV = Phi
do c = 1, NCells_PE(Mesh)
  X = Coordinates_Cells(1,c)
  Exact(c) = (Q0 / six) * ((one + eight * D0)/(one + four*D0) * &
             (one + X / (two*D0)) - X**4 / (two*D0))
! write (6,*) c, Coordinates_Cells(1,c), Phi(c), Exact(c), Exact(c) - Phi(c)
end do
Error_MV = Phi - Exact
call Output (Error_MV, 0, 0)
Exact_MV = Exact
Relative_L2_Error_Norm = Norm(Error_MV) / Norm(Exact_MV)
if (this_is_IO_PE) &
  write (6,102) '  Relative L2 error norm = ', Relative_L2_Error_Norm

! Plotting output (turn on if desired).

if (.false.) then
  call Dump_GMV ('GMVout.gmv', Mesh, Phi_MV, Exact_MV, Error_MV)
end if
```

```
! Check gradient.

call Evaluate_Gradient_Cells (Ortho_Diff_Term, Phi_MV, Gradient_Cells)
do c = 1, NCells_PE(Mesh)
  X = Coordinates_Cells(1,c)
  if (ABS(Gradient_Cells(1,c) - &
          ((Q0 / six) * ((one + eight * D0)/(one + four*D0) * &
          (one / (two*D0)) - four*X**3 / (two*D0))) ) > 0.12d0 .or. &
      ABS(Gradient_Cells(2,c) - (zero) ) > 1.d-8 .or. &
      ABS(Gradient_Cells(3,c) - (zero) ) > 1.d-8) then
    write (6,*) 'Error (Problem X.6): Grad = ', Gradient_Cells(:,c)
    write (6,*) 'Error (Problem X.6): Diff = ', ABS(Gradient_Cells(1,c) - &
          ((Q0 / six) * ((one + eight * D0)/(one + four*D0) * &
          (one / (two*D0)) - four*X**3 / (two*D0))))
  end if
end do

! Finalize terms.

call Finalize (Ortho_Diff_Term)
call Finalize (Linear_Term)
call Finalize (Constant_Term)

! Reset entire Matrix Equation.
!
! Note that we must finalize X_MV before Matrix_ELLM because the call to
! Solver may use a MatVec to calculate a residual, which would create an
! OV in X_MV based on the Data_Index in Matrix_ELLM. Tricky.

call Finalize (X_MV)
call Finalize (RHS_MV)
call Finalize (Matrix_ELLM)
call Initialize (Matrix_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                 'Coefficients', status)
call Initialize (RHS_MV, Row_Structure, 'Right-Hand Side', &
                 status)
call Initialize (X_MV, Column_Structure, 'Solution', status)

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
! Problem X.7: 1-D Quartic Solution with Two Materials
!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

if (this_is_IO_PE) then
  write (6,100) 'Problem X.7: 1-D Quartic Solution with Two Materials'
end if

! Problem parameters.

D1 = one/3.d0
D2 = one/3.d6
Q0 = 0.00023d0

! Set Absorption (Removal) cross-section for each cell.
```

```
Sigma_a = zero

! Set Source term for each cell.

do c = 1, NCells_PE(Mesh)
  Source(c) = -Q0 * Coordinates_Cells(1,c)**2
end do

! Set Diffusion Coefficient for each cell.

where (Coordinates_Cells(1,:) < 0.5)
  Diffusion_Coefficient(:) = D1
elsewhere
  Diffusion_Coefficient(:) = D2
end where

! Set Boundary Conditions.

where (Flag_Faces_of_Cells == 1)     ! Left (-x) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 2) ! Right (+x) Face.
  BC_Faces_of_Cells = Vacuum_BC
elsewhere (Flag_Faces_of_Cells == 3) ! Front (-y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 4) ! Back (+y) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 5) ! Bottom (-z) Face.
  BC_Faces_of_Cells = Reflective_BC
elsewhere (Flag_Faces_of_Cells == 6) ! Top (+z) Face.
  BC_Faces_of_Cells = Reflective_BC
end where
Phi_BC_Faces_of_Cells = zero

! Set Phi and X to an initial guess to avoid division by zero error in
! LAMG in the stopping test.

Phi = one
Phi_MV = Phi
X_BNV = one
X_MV = X_BNV

! Define Steady State Diffusion Equation.

call Initialize (Ortho_Diff_Term, Diffusion_Coefficient, BC_Faces_of_Cells, &
                 Phi_BC_Faces_of_Cells, Phi_MV, 'Cells', Mesh, &
                 'Diffusion Term', Extrapolation, Equation, NEquations, &
                 status)
Exponent = one
call Initialize (Linear_Term, Sigma_a, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Absorption Term', Derivative, Delta_t, Phi_MV, Variable, &
                 Equation, NEquations, status)
Exponent = zero
call Initialize (Constant_Term, Source, Exponent, Phi_MV, 'Cells', Mesh, &
                 'Source Term', Derivative, Delta_t, Phi_MV, Variable, &
```

```
                 Equation, NEquations, status)

  ! Output Terms.

  NRows = NCells_PE(Mesh)
  !call Output ( Constant_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  !call Output (   Linear_Term,  MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  !call Output (Ortho_Diff_Term, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Check state of Equation Terms.

  VERIFY(Valid_State(  Constant_Term),0)
  VERIFY(Valid_State(    Linear_Term),0)
  VERIFY(Valid_State(Ortho_Diff_Term),0)

  ! Populate Matrix Equation.

  call Add_to_Matrix_Equation (Ortho_Diff_Term, Matrix_ELLM, RHS_MV)
  call Add_to_Matrix_Equation (Linear_Term, Matrix_ELLM, RHS_MV)
  call Add_to_Matrix_Equation (Constant_Term, Matrix_ELLM, RHS_MV)

  ! ------------------------------------------------------------------------
  ! This is a difficult problem, and it turned out that setting the diagonals
  ! to unity on zero rows, as is done for the other problems, resulted in a
  ! fairly good answer, but the gradients didn't pass the current tests. So,
  ! instead of taking that approach, we enter the same equation multiple times.
  ! So we repeat the steps above for the other equations. This works well.

  do eq = 1, NEquations
    if (eq /= Equation) then

      ! Finalize terms.

      call Finalize (Ortho_Diff_Term)
      call Finalize (Linear_Term)
      call Finalize (Constant_Term)

      ! Define Steady State Diffusion Equation for Equation = eq.

      call Initialize (Ortho_Diff_Term, Diffusion_Coefficient, &
                       BC_Faces_of_Cells, Phi_BC_Faces_of_Cells, Phi_MV, &
                       'Cells', Mesh, 'Diffusion Term', Extrapolation, eq, &
                       NEquations, status)
      Exponent = one
      call Initialize (Linear_Term, Sigma_a, Exponent, Phi_MV, 'Cells', Mesh, &
                       'Absorption Term', Derivative, Delta_t, Phi_MV, &
                       Variable, eq, NEquations, status)
      Exponent = zero
      call Initialize (Constant_Term, Source, Exponent, Phi_MV, 'Cells', Mesh, &
                       'Source Term', Derivative, Delta_t, Phi_MV, &
                       Variable, eq, NEquations, status)

      ! Populate Matrix Equation for Equation = eq.
```

```
      call Add_to_Matrix_Equation (Ortho_Diff_Term, Matrix_ELLM, RHS_MV)
      call Add_to_Matrix_Equation (Linear_Term, Matrix_ELLM, RHS_MV)
      call Add_to_Matrix_Equation (Constant_Term, Matrix_ELLM, RHS_MV)

    end if
  end do

  ! End of repeated equation.
  ! ------------------------

  ! Solve Matrix Equation.

  VERIFY(Valid_State(Matrix_ELLM),0)
  !call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  call Initialize (Solver, 'LAMG')
  call Set (Solver, 'Epsilon', 1.d-10)
  call Solve (Solver, Matrix_ELLM, X_MV, RHS_MV)
  call Finalize (Solver)

  ! Output.

  VERIFY(Valid_State(X_MV),0)
  !call Output (X_MV)
  !call Output (Matrix_ELLM, MAX(1, NRows/10), MIN(NRows, NRows/10+50))
  VERIFY(Valid_State(RHS_MV),0)
  !call Output (RHS_MV, MAX(1, NRows/10), MIN(NRows, NRows/10+50))

  ! Check answers.
  ! Note: I tested problem X.7 and it converged by 2nd Order when
  ! refined in the X-direction.

  X_BNV = X_MV
  Phi = X_BNV(Equation::NEquations)
  Phi_MV = Phi
  do c = 1, NCells_PE(Mesh)
    X = Coordinates_Cells(1,c)
    if (X < 0.5) then
      Exact(c) = Q0 / (twelve*D2) * (one + eight * D2 + &
                 (D2 - D1) / (sixteen*D1) - (D2/D1) * X**4)
    else
      Exact(c) = Q0 / (twelve*D2) * (one + eight * D2 - X**4)
    end if
   ! write (6,*) c, Coordinates_Cells(1,c), Phi(c), Exact(c), Exact(c) - Phi(c)
  end do
  Error_MV = Phi - Exact
  call Output (Error_MV, 0, 0)
  Exact_MV = Exact
  Relative_L2_Error_Norm = Norm(Error_MV) / Norm(Exact_MV)
  if (this_is_IO_PE) &
    write (6,102) '  Relative L2 error norm = ', Relative_L2_Error_Norm

  ! Plotting output (turn on if desired).

  if (.false.) then
```

```
    call Dump_GMV ('GMVout.gmv', Mesh, Phi_MV, Exact_MV, Error_MV)
  end if

  ! Check gradient.

  call Evaluate_Gradient_Cells (Ortho_Diff_Term, Phi_MV, Gradient_Cells)
  do c = 1, NCells_PE(Mesh)
    X = Coordinates_Cells(1,c)
    if (X < 0.5) then
      if (ABS(Gradient_Cells(1,c) - &
              (-Q0 / (three*D1) * X**3) ) > 1.1d0 .or. &
          ABS(Gradient_Cells(2,c) - (zero) ) > 1.d-8 .or. &
          ABS(Gradient_Cells(3,c) - (zero) ) > 1.d-8) then
        write (6,*) 'Error (Problem X.7): Grad = ', Gradient_Cells(:,c)
        write (6,*) 'Error (Problem X.7): Real = ', &
              (-Q0 / (three*D1) * X**3), X
        write (6,*) 'Error (Problem X.7): Diff = ', &
              ABS(Gradient_Cells(1,c) - (-Q0 / (three*D1) * X**3) )
      end if
    else
      if (ABS(Gradient_Cells(1,c) - &
              (-Q0 / (three*D2) * X**3) ) > 1.1d0 .or. &
          ABS(Gradient_Cells(2,c) - (zero) ) > 1.d-8 .or. &
          ABS(Gradient_Cells(3,c) - (zero) ) > 1.d-8) then
        write (6,*) 'Error (Problem X.7): Grad = ', Gradient_Cells(:,c)
        write (6,*) 'Error (Problem X.7): Real = ', &
              (-Q0 / (three*D2) * X**3), X
        write (6,*) 'Error (Problem X.7): Diff = ', &
              ABS(Gradient_Cells(1,c) - (-Q0 / (three*D2) * X**3) )
      end if
    end if
  end do

  ! Finalize terms.

  call Finalize (Ortho_Diff_Term)
  call Finalize (Linear_Term)
  call Finalize (Constant_Term)

  ! Reset entire Matrix Equation.
  !
  ! Note that we must finalize X_MV before Matrix_ELLM because the call to
  ! Solver may use a MatVec to calculate a residual, which would create an
  ! OV in X_MV based on the Data_Index in Matrix_ELLM. Tricky.

  call Finalize (X_MV)
  call Finalize (RHS_MV)
  call Finalize (Matrix_ELLM)
  call Initialize (Matrix_ELLM, MaxNonzeros, Row_Structure, Column_Structure, &
                   'Coefficients', status)
  call Initialize (RHS_MV, Row_Structure, 'Right-Hand Side', &
                   status)
  call Initialize (X_MV, Column_Structure, 'Solution', status)
```

```
!~~~~~~~~~~~End of test problems.~~~~~~~~~~~

  ! Finalize variables.

  call Finalize (BC_Faces_of_Cells)
  call Finalize (Flag_Faces_of_Cells)
  call Finalize (Phi_BC_Faces_of_Cells)
  call Finalize (Coordinates_Faces_of_Cells)
  call Finalize (MaxNonzeros)
  call Finalize (Phi_MV)
  call Finalize (Error_MV)
  call Finalize (RHS_MV)
  call Finalize (Matrix_ELLM)
  call Finalize (Phi)
  call Finalize (Source)
  call Finalize (Sigma_a)
  call Finalize (Diffusion_Coefficient)
  call Finalize (Mesh)
  call Finalize (NDimensions)
  call Finalize (Lengths)
  call Finalize (NCells_X)
  call Finalize (NCells_Y)
  call Finalize (NCells_Z)
  call Finalize (Comm)

  ! Format statements.

  100 format (/,a,:,a,:,a,:,a)
  102 format (a,1pe22.15)

end
```

Appendix I

Mesh Module Code Listing

The main documentation of the Mesh Module in chapter 14 on page 169 contains additional explanation of this code listing.

I.1 Multi_Mesh Class Code Listing

The main documentation of the Multi_Mesh Class in § 14.1 on page 169 contains additional explanation of this code listing.

```
!
!     Author: Michael L. Hall
!             P.O. Box 1663, MS-D413, LANL
!             Los Alamos, NM 87545
!             ph: 505-665-4312
!             email: Hall@LANL.gov
!
!     Created on: 04/20/00
!     CVS Info:   $Id: multi_mesh.F90,v 1.39 2009/09/17 01:11:51 hall Exp $

module Caesar_Multi_Mesh_Class

  ! Global use associations.

  use Caesar_Data_Structures_Module
  use Caesar_Numbers_Module, only: zero, fourth, half, one
  ! Only used for GMV dumps of MVs.
  use Caesar_Linear_Algebra_Module

  ! Start up with everything untyped and private.

  implicit none
  private

  ! Public procedures.

  public :: Initialize, Finalize, Valid_State, Initialized
  public :: Assignment (=), Cell_Structure, Dump_GMV, Dump_XMGrace, &
            Face_Structure, First_Cell_PE, First_Face_PE, First_Node_PE, &
```

```
           Get_Area_Faces_of_Cells, Get_Faces_per_Cell, &
           Get_Coordinates_Cells, Get_Coordinates_Cells_of_Cells, &
           Get_Coordinates_Faces_of_Cells, Get_Coordinates_Nodes_of_Cells, &
           Get_DeltaR1f_Cells_of_Cells, Get_DeltaR2f_Cells_of_Cells, &
           Get_DeltaR21_Cells_of_Cells, Get_Flag_Faces_of_Cells, &
           Get_NDimensions, Get_Volume_Cells, Last_Cell_PE, Last_Face_PE, &
           Last_Node_PE, Name, NCells_PE, NCells_Total, NFaces_PE, &
           NFaces_Total, NNodes_PE, NNodes_Total, Node_Structure, &
           Output, Range_Cells_PE, Range_Faces_PE, Range_Nodes_PE, &
           Set_Version, Version

  interface Initialize
    module procedure Initialize_Uniform_Multi_Mesh
    module procedure Initialize_Orthogonal_MMesh
    module procedure Initialize_Base_Multi_Mesh
  end interface

  interface Finalize
    module procedure Finalize_Multi_Mesh
  end interface

  interface Valid_State
    module procedure Valid_State_Multi_Mesh
  end interface

  interface Initialized
    module procedure Initialized_Multi_Mesh
  end interface

!  interface Dump_CGNS
!    module procedure Dump_CGNS_Multi_Mesh
!  end interface

  interface Dump_GMV
    module procedure Dump_GMV_Multi_Mesh
  end interface

  interface Dump_XMGrace
    module procedure Dump_XMGrace_Multi_Mesh
  end interface

  fortext([Value],
          [Name Cell_Structure Node_Structure Face_Structure dnl
           First_Cell_PE Last_Cell_PE NCells_PE NCells_Total dnl
           First_Node_PE Last_Node_PE NNodes_PE NNodes_Total dnl
           First_Face_PE Last_Face_PE NFaces_PE NFaces_Total dnl
           Range_Cells_PE Range_Nodes_PE Range_Faces_PE],[
    interface Value
      module procedure expand(Get_Value_Multi_Mesh)
    end interface
  ])

  interface Get_Area_Faces_of_Cells
    module procedure Get_Area_Faces_of_Cells_MMesh
```

```
  end interface

  interface Get_Faces_per_Cell
    module procedure Get_Faces_per_Cell_Multi_Mesh
  end interface

  interface Get_Flag_Faces_of_Cells
    module procedure Get_Flag_Faces_of_Cells_MMesh
  end interface

  interface Get_Coordinates_Cells
    module procedure Get_Coordinates_Cells_MMesh
  end interface

  interface Get_Coordinates_Cells_of_Cells
    module procedure Get_Coordinates_CoC_MMesh
  end interface

  interface Get_Coordinates_Faces_of_Cells
    module procedure Get_Coordinates_FoC_MMesh
  end interface

  interface Get_Coordinates_Nodes_of_Cells
    module procedure Get_Coordinates_NoC_MMesh
  end interface

  interface Get_DeltaR1f_Cells_of_Cells
    module procedure Get_DeltaR1f_C_of_C_MMesh
  end interface

  interface Get_DeltaR2f_Cells_of_Cells
    module procedure Get_DeltaR2f_C_of_C_MMesh
  end interface

  interface Get_DeltaR21_Cells_of_Cells
    module procedure Get_DeltaR21_C_of_C_MMesh
  end interface

  interface Get_NDimensions
    module procedure Get_NDimensions_Multi_Mesh
  end interface

  interface Get_Volume_Cells
    module procedure Get_Volume_Cells_Multi_Mesh
  end interface

!  interface Set_Coordinates
!    module procedure Set_Coordinates_Multi_Mesh_1
!    module procedure Set_Coordinates_Multi_Mesh_2
!    module procedure Set_Coordinates_Multi_Mesh_3
!    module procedure Set_Coordinates_Multi_Mesh_4
!    module procedure Set_Coordinates_Multi_Mesh_5
!  end interface
```

```
interface Set_Version
  module procedure Set_Version_Multi_Mesh
end interface

interface Version
  module procedure Get_Version_Multi_Mesh
end interface

! Public type definitions.

public :: Multi_Mesh_type

type Multi_Mesh_type

  ! Initialization flag.

  type(integer) :: Initialized

  ! The name for this mesh.

  type(character,name_length) :: Name

  ! Version number which is incremented every time the mesh is modified.

  type(integer) :: Version

  ! Mesh type information.

  type(integer)               :: NDimensions   ! Number of Dimensions.
  type(character,name_length) :: Geometry      ! Cartesian (x, xy, or xyz),
                                               ! Cylindrical (r or rz), or
                                               ! Spherical (r)
  type(character,name_length) :: Uniformity    ! "Uniform" or "Nonuniform"
  type(character,name_length) :: Orthogonality ! "Orthogonal" or
                                               ! "Nonorthogonal"
  type(character,name_length) :: Structure     ! "Structured" or
                                               ! "Unstructured"
  type(logical)               :: AMR           ! Adaptive Mesh Refinement
                                               ! (H-type).
  type(character,name_length) :: Shape         ! "Segmented", "Triangular",
                                               ! "Quadrilateral",
                                               ! "Polygonal", "Tetrahedral",
                                               ! "Hexahedral" or "Polyhedral"

  ! Base structures for the mesh.

  type(Base_Structure_type) ::  Node_Structure
  type(Base_Structure_type) ::  Cell_Structure
  type(Base_Structure_type) ::  Face_Structure
  type(Base_Structure_type) ::  Boundary_Face_Structure

  ! Connectivity information for the mesh.

  type(Data_Index_type) :: Nodes_of_Cells_Index
```

```
type(Data_Index_type) :: Nodes_of_Faces_Index
type(Data_Index_type) :: Faces_of_Cells_Index
type(Data_Index_type) :: Cells_of_Faces_Index
! May depend on discretization.
type(Data_Index_type) :: Faces_of_Faces_Index
type(Data_Index_type) :: Cells_of_Cells_Index

! Coordinate data.

type(Distributed_Vector_type) :: Coordinates_Nodes_DV
type(Overlapped_Vector_type)  :: Coordinates_Nodes_of_Cells_OV
type(Collected_Array_type)    :: Coordinates_Nodes_of_Cells_CA
type(Overlapped_Vector_type)  :: Coordinates_Nodes_of_Faces_OV
type(Collected_Array_type)    :: Coordinates_Nodes_of_Faces_CA

! Flags.

type(Distributed_Vector_type) :: Flags_Nodes_DV
type(Distributed_Vector_type) :: Flags_Cells_DV
type(Distributed_Vector_type) :: Flags_Faces_DV

! User-numbering, that is, numbers assigned by the user that may
! be different from those needed by the mesh class itself.

type(Distributed_Vector_type) :: User_Numbers_Nodes_DV
type(Distributed_Vector_type) :: User_Numbers_Cells_DV
type(Distributed_Vector_type) :: User_Numbers_Faces_DV

! Point locations (calculated).

type(Distributed_Vector_type) :: Coordinates_Cells_DV
type(Distributed_Vector_type) :: Coordinates_Faces_DV
type(Collected_Array_type)    :: Coordinates_Faces_of_Cells_CA
type(Collected_Array_type)    :: Coordinates_Cells_of_Cells_CA
type(Collected_Array_type)    :: QuarterPoints_Faces_of_Cells_CA

! Volume and area data.

type(Distributed_Vector_type) :: Volume_Cells_DV
type(Collected_Array_type)    :: Vector_Area_Faces_of_Cells_CA
type(Collected_Array_type)    :: Unit_Normal_Faces_of_Cells_CA
type(Collected_Array_type)    :: Area_Faces_of_Cells_CA
type(Distributed_Vector_type) :: Area_Faces_DV
type(Distributed_Vector_type) :: JMT_Cells_DV

! Mesh scalar info.

type(integer) :: NCells_total   ! Number of Cells in the whole problem.
type(integer) :: NCells_PE      ! Number of Cells on this PE.
type(integer) :: First_Cell_PE  ! First global cell number on this PE.
type(integer) :: Last_Cell_PE   ! Last global cell number on this PE.
type(integer), dimension(2) :: Range_Cells_PE ! Cell number range for PE.

type(integer) :: NNodes_total   ! Number of Nodes in the whole problem.
```

```
    type(integer) :: NNodes_PE      ! Number of Nodes on this PE.
    type(integer) :: First_Node_PE  ! First global node number on this PE.
    type(integer) :: Last_Node_PE   ! Last global node number on this PE.
    type(integer), dimension(2) :: Range_Nodes_PE ! Node number range for PE.

    type(integer) :: NFaces_total   ! Number of Faces in the whole problem.
    type(integer) :: NFaces_PE      ! Number of Faces on this PE.
    type(integer) :: First_Face_PE  ! First global face number on this PE.
    type(integer) :: Last_Face_PE   ! Last global face number on this PE.
    type(integer), dimension(2) :: Range_Faces_PE ! Face number range for PE.

    type(integer) :: Faces_per_Cell ! = 2*NDimensions for Type 2 mesh.
    type(integer) :: Nodes_per_Cell ! = 2**NDimensions for Type 2 mesh.
    type(integer) :: Nodes_per_Face ! = 2**(NDimensions-1) for Type 2 mesh.

    ! Global mesh data (valid for uniform meshes only).

    type(real)   :: Volume_All_Cells
    type(real,1) :: Area_All_Faces ! Scalar face area, stored as a vector for
                                   ! the number of dimensions.
    ! Vector of physical mesh extents in each direction.
    type(real,1) :: Lengths

    ! Mesh edge data (valid for orthogonal meshes only).

    type(real,1) :: Coordinates_Nodes_X, & ! Coordinate vectors of the nodes
                    Coordinates_Nodes_Y, & ! along an edge of the mesh, stored
                    Coordinates_Nodes_Z    ! on all PEs.

    ! Face Flags for Structured Meshes. This is set to: 0 for internal,
    ! 1 for left (-x), 2 for right (+x), 3 for front (-y), 4 for back (+y),
    ! 5 for bottom (-z), 6 for top (+z).

    type(integer,2) :: Flag_Faces_of_Cells

  end type Multi_Mesh_type

!Operations:
!  call Initialize (Mesh,Mesh_type,Geometry,Coordinates,NNodes_PE,etc.)
!  call Initialize (Mesh,File)    ! Synonym = Read_from_File
!  call Initialize (Mesh,Mesh_type,"Kershaw (or other mesh type)")
!                  "Uniform", "Random", "Kershaw", "Kershaw-squared", "Shestakov"
!  call Finalize (Mesh)
!  logical = Valid_State(Mesh)
!  Node_Structure => Get_Node_Structure(Mesh)   ! returns a pointer.
!  Cell_Structure => Get_Cell_Structure(Mesh)   ! returns a pointer.
!  Face_Structure => Get_Face_Structure(Mesh)   ! returns a pointer.
!  Boundary_Face_Structure => Get_Boundary_Face_Structure(Mesh)
!    --> Ditto for all the Data_Index variables.
!  Get_Vector_Area_Faces_of_Cells (Mesh)
!  Get_Area_Faces (Mesh)
!  Get_Volume_Cells (Mesh)
!  Get_JMT_Cells (Mesh)
!  call Set_Coordinates (Mesh, Coordinates_BNV)   ! Synonym = Move
```

```
!  Coordinates_Nodes_BNV = Get_Coordinates_Nodes(Mesh)
!  Coordinates_Cells_BNV = Get_Coordinates_Cells(Mesh)
!  Coordinates_Faces_BNV = Get_Coordinates_Faces(Mesh)
!  Coordinates_Faces_of_Cells_BNV = Get_Coordinates_Faces_of_Cells(Mesh)
!  QuarterPoints_BNV = Get_QuarterPoints(Mesh)
!  Functions to return the mesh scalar variables,
!    i.e. integer = NDimensions(Mesh), logical = AMR(Mesh),
!    logical = Uniform(Mesh), etc.
!  From draco''s AMR_Mesh... useful?
!    min(Mesh) - Returns the minimum coordinate value along the
!      specified direction for the specified cell. Ditto, max.
!    get_cell (Mesh) Returns the cell number that contains the
!      specified point in space.
!    get_db (Mesh, Omega) Returns the distance to the nearest
!      boundary in the specified cell and direction for the
!      specified point in space.
!    get_generation (Mesh) Returns the generation (i.e., refinement)
!      level for the specified cell.
!  Get_Cell_Flags (Mesh, DV)
!  Get_Face_Flags (Mesh, DV)
!  Get_Node_Flags (Mesh, DV)
!  Get_Boundary_Face_Flags (Mesh, DV)
!  call Output (Mesh)
!  call Output (Mesh,"Format (like GMV, CGNS, RTT)")
!  call Output (Mesh, DV(:), "Format (like GMV, CGNS, RTT)")
!  call Interpolate (Mesh, DV1, DV2, Interpolation_type)

  ! Global class variables.

  type(integer), parameter :: Version_Increment = 1
  type(character,name_length), dimension(3), parameter :: &
    Geometry_Options = (/"Cartesian  ", "Cylindrical", "Spherical  "/)
  type(character,name_length), dimension(2), parameter :: &
    Uniformity_Options =  (/"Uniform   ", "Nonuniform"/)
  type(character,name_length), dimension(2), parameter :: &
    Orthogonality_Options =  (/"Orthogonal   ", "Nonorthogonal"/)
  type(character,name_length), dimension(2), parameter :: &
    Structure_Options =  (/"Structured  ", "Unstructured"/)
  type(character,name_length), dimension(7), parameter :: &
    Shape_Options = (/"Segmented    ", "Triangular   ", "Quadrilateral", &
                      "Polygonal    ", "Tetrahedral  ", "Hexahedral   ", &
                      "Polyhedral   "/)

contains
```

The Multi_Mesh Class contains the following routines which are listed in separate sections:

end module Caesar_Multi_Mesh_Class

I.1.1 Initialize_Base_Multi_Mesh Procedure

The main documentation of the Initialize_Base_Multi_Mesh Procedure in § 14.1.1 on page 172 contains additional explanation of this code listing.

```
  subroutine Initialize_Base_Multi_Mesh (Mesh, NDimensions, Geometry, &
    Uniformity, Orthogonality, Structure, AMR, Shape, &
    NNodes_Vector, NCells_Vector, NFaces_Vector, &
    Coordinates_Nodes_PE, Nodes_of_Cells_PE, Mesh_Name, status)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag
```

```
! ~~~~~~~~~~~~~~~~
! Input variables.
! ~~~~~~~~~~~~~~~~

! Mesh type information.

type(integer), intent(in)     :: NDimensions    ! Number of Dimensions.
type(character,*), intent(in) :: Geometry       ! Mesh geometry.
type(character,*), intent(in) :: Uniformity     ! Uniform or Nonuniform.
type(character,*), intent(in) :: Orthogonality  ! Orthogonal or
                                                ! Nonorthogonal.
type(character,*), intent(in) :: Structure      ! Structured or
                                                ! Unstructured.
type(logical), intent(in)     :: AMR            ! Adaptive Mesh Refinement.
type(character,*), intent(in) :: Shape          ! Cell shape.

! Structure length vectors, which give numbers for all PEs.

type(integer,1) :: NNodes_Vector ! Number of nodes.
type(integer,1) :: NCells_Vector ! Number of cells.
type(integer,1) :: NFaces_Vector ! Number of faces.

! Mesh coordinates and indices.

! The coordinates of the nodes on this PE.
type(real,2) :: Coordinates_Nodes_PE
! The nodes for the cells on this PE.
type(integer,2) :: Nodes_of_Cells_PE

type(character,*), intent(in), optional :: Mesh_Name ! Mesh name.

! ~~~~~~~~~~~~~~~~~
! Output variables.
! ~~~~~~~~~~~~~~~~~

! Multi_Mesh to be initialized.
type(Multi_Mesh_type), intent(inout) :: Mesh
type(Status_type), intent(out), optional :: status  ! Exit status.

! ~~~~~~~~~~~~~~~~~~~
! Internal variables.
! ~~~~~~~~~~~~~~~~~~~

type(Status_type), dimension(20) :: allocate_status ! Allocation Status.
type(Status_type) :: consolidated_status           ! Consolidated Status.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

! Mesh type info is valid.
VERIFY(NDimensions .InInterval. (/1, 3/),5)
VERIFY(Geometry .InSet. Geometry_Options,5)
VERIFY(Uniformity .InSet. Uniformity_Options,5)
```

```
VERIFY(Orthogonality .InSet. Orthogonality_Options,5)
VERIFY(Structure .InSet. Structure_Options,5)
VERIFY(Valid_State(AMR),5)
VERIFY(Shape .InSet. Shape_Options,5)
! Mesh length vectors are correct length.
VERIFY(SIZE(NNodes_Vector) == NPEs,5)
VERIFY(SIZE(NCells_Vector) == NPEs,5)
VERIFY(SIZE(NFaces_Vector) == NPEs,5)
! Input arrays are correct size.
VERIFY(SIZE(Coordinates_Nodes_PE,1) == NDimensions,5)
VERIFY(SIZE(Coordinates_Nodes_PE,2) == NNodes_Vector(this_PE),5)
VERIFY(SIZE(Nodes_of_Cells_PE,1) == NCells_Vector(this_PE),5)
VERIFY(SIZE(Nodes_of_Cells_PE,2) == 2**NDimensions,5)

! Set up internals.

if (PRESENT(Mesh_Name)) Mesh%Name = Mesh_Name

! Set Mesh type information.

Mesh%NDimensions   = NDimensions
Mesh%Geometry      = Geometry
Mesh%Uniformity    = Uniformity
Mesh%Orthogonality = Orthogonality
Mesh%Structure     = Structure
Mesh%AMR           = AMR
Mesh%Shape         = Shape

! Mesh scalar info.

Mesh%NCells_total   = SUM(NCells_Vector)
Mesh%NCells_PE      = NCells_Vector(this_PE)
Mesh%Last_Cell_PE   = SUM(NCells_Vector(1:this_PE))
Mesh%First_Cell_PE  = Mesh%Last_Cell_PE - Mesh%NCells_PE + 1
Mesh%Range_Cells_PE = (/ Mesh%First_Cell_PE, Mesh%Last_Cell_PE /)

Mesh%NNodes_total   = SUM(NNodes_Vector)
Mesh%NNodes_PE      = NNodes_Vector(this_PE)
Mesh%Last_Node_PE   = SUM(NNodes_Vector(1:this_PE))
Mesh%First_Node_PE  = Mesh%Last_Node_PE - Mesh%NNodes_PE + 1
Mesh%Range_Nodes_PE = (/ Mesh%First_Node_PE, Mesh%Last_Node_PE /)

Mesh%NFaces_total   = SUM(NFaces_Vector)
Mesh%NFaces_PE      = NFaces_Vector(this_PE)
Mesh%Last_Face_PE   = SUM(NFaces_Vector(1:this_PE))
Mesh%First_Face_PE  = Mesh%Last_Face_PE - Mesh%NFaces_PE + 1
Mesh%Range_Faces_PE = (/ Mesh%First_Face_PE, Mesh%Last_Face_PE /)

select case (Mesh%Shape)
! Type 1 meshes.
case ('Tetrahedral', 'Triangular')
  Mesh%Nodes_per_Cell = NDimensions+1
  Mesh%Nodes_per_Face = NDimensions
  Mesh%Faces_per_Cell = NDimensions+1
```

```
    ! Type 2 meshes:
    case ('Segmented', 'Quadrilateral', 'Hexahedral')
      Mesh%Nodes_per_Cell = 2**NDimensions
      Mesh%Nodes_per_Face = 2**(NDimensions-1)
      Mesh%Faces_per_Cell = 2*NDimensions
    ! Type N meshes.
    case ('Polygonal', 'Polyhedral')
      Mesh%Nodes_per_Cell = 0
      Mesh%Nodes_per_Face = 0
      Mesh%Faces_per_Cell = 0
    end select
    ! Segmented is either Type 1 or Type 2.

    ! Allocations and initializations.

    call Initialize (allocate_status)
    call Initialize (consolidated_status)

    ! Set up mesh structures.

    call Initialize (Mesh%Node_Structure, NNodes_Vector, 'Nodes', &
                     allocate_status(1))
    call Initialize (Mesh%Cell_Structure, NCells_Vector, 'Cells', &
                     allocate_status(2))
    call Initialize (Mesh%Face_Structure, NFaces_Vector, 'Faces', &
                     allocate_status(3))

    ! Set Mesh coordinates.

    call Initialize (Mesh%Coordinates_Nodes_DV, Mesh%Node_Structure, &
                     2, 'Coordinates of Nodes', status, NDimensions)
    Mesh%Coordinates_Nodes_DV = Coordinates_Nodes_PE

    ! Set Mesh indices.

    call Initialize (Mesh%Nodes_of_Cells_Index, Mesh%Node_Structure, &
                     Mesh%Cell_Structure, &
                     Many_of_One_Array=Nodes_of_Cells_PE, &
                     status=allocate_status(4))

    ! Consolidate and handle status.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

    ! Set initialization flag.
```

```
    Mesh%Initialized = initialized_flag

    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)  ! Mesh is now valid.

    return
  end subroutine Initialize_Base_Multi_Mesh
```

I.1.2 Initialize_Uniform_Multi_Mesh Procedure

The main documentation of the Initialize_Uniform_Multi_Mesh Procedure in § 14.1.2 on page 173 contains additional explanation of this code listing.

```
  subroutine Initialize_Uniform_Multi_Mesh (Mesh, NDimensions, Lengths, &
                                            NCells_X_total, NCells_Y_total, &
                                            NCells_Z_total, Mesh_Name, status)

    ! Input variables.

    type(integer), intent(in) :: NDimensions    ! Number of Dimensions.
    type(real,1,np), intent(in) :: Lengths      ! Physical extent of the
                                                ! domain in each direction.
    ! Total number of cells in the X-, Y-, and Z-directions.
    type(integer), intent(in)              :: NCells_X_total
    type(integer), intent(inout), optional :: NCells_Y_total
    type(integer), intent(inout), optional :: NCells_Z_total
    type(character,*), intent(in), optional :: Mesh_Name ! Mesh name.

    ! Output variables.

    ! Multi_Mesh to be initialized.
    type(Multi_Mesh_type), intent(inout) :: Mesh
    type(Status_type), intent(out), optional :: status  ! Exit status.

    ! Internal variables.

    type(character,name_length) :: Shape         ! Cell shape.
    type(character,name_length) :: Geometry      ! Cell geometry (Cartesian).
    type(character,name_length) :: Uniformity    ! Set to "Uniform".
    type(character,name_length) :: Orthogonality ! Set to "Orthogonal".
    type(character,name_length) :: Structure     ! Set to "Structured".
    type(logical)               :: AMR           ! Set to false.
    type(Status_type), dimension(20) :: allocate_status  ! Allocation Status.
    type(Status_type) :: consolidated_status             ! Consolidated Status.
    type(integer) :: NPEs_X ! Number of PEs in X.
    type(integer) :: NPEs_Y ! Number of PEs in Y.
    type(integer) :: NPEs_Z ! Number of PEs in Z.

    ! Structure length vectors, which give numbers for all PEs.
    type(integer,1) :: NCells_Vector   ! Number of cells.
    type(integer,1) :: NCells_X_Vector ! Number of cells in the X direction.
```

```
type(integer,1) :: NCells_Y_Vector ! Number of cells in the Y direction.
type(integer,1) :: NCells_Z_Vector ! Number of cells in the Z direction.
type(integer,1) :: NFaces_Vector   ! Number of faces.
type(integer,1) :: NNodes_Vector   ! Number of nodes.
type(integer,1) :: NNodes_X_Vector ! Number of nodes in the X direction.
type(integer,1) :: NNodes_Y_Vector ! Number of nodes in the Y direction.
type(integer,1) :: NNodes_Z_Vector ! Number of nodes in the Z direction.

! Location of this_PE in the PE-mesh.
type(integer) :: this_PE_X, this_PE_Y, this_PE_Z
type(integer) :: node, pe_x, pe_y, pe_z     ! Loop parameters.

! Offsets - starting points for this PE.
type(real) :: Offset_PE_X, Offset_PE_Y, Offset_PE_Z ! 1st node coordinates.

! Mesh coordinates and indices.

! The coordinates of the nodes on this PE.
type(real,2) :: Coordinates_Nodes_PE
! The nodes for the cells on this PE.
type(integer,2) :: Nodes_of_Cells_PE
! The cells for the cells (that is, across each face) on this PE.
type(integer,2) :: Cells_of_Cells_PE
! Face Flags for Structured Meshes.
type(integer,2) :: Flag_Faces_of_Cells

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

! Mesh type info is valid.
VERIFY(NDimensions .InInterval. (/1, 3/),5)
VERIFY(SIZE(Lengths) == NDimensions,5)   ! Lengths is correct size.

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)

! Generate the connectivity.

call Gen_StructureMesh_Connectivity (NDimensions, Lengths, &
  NCells_X_total, NCells_Y_total, NCells_Z_total, Shape, Structure, AMR, &
  NPEs_X, NPEs_Y, NPEs_Z, this_PE_X, this_PE_Y, this_PE_Z, &
  NNodes_Vector, NCells_Vector, NFaces_Vector, &
  NCells_X_Vector, NCells_Y_Vector, NCells_Z_Vector, &
  NNodes_X_Vector, NNodes_Y_Vector, NNodes_Z_Vector, &
  Nodes_of_Cells_PE, Cells_of_Cells_PE, Flag_Faces_of_Cells, &
  allocate_status(1))

! Set mesh type info for a Uniform Mesh.

Geometry = 'Cartesian'
Uniformity = 'Uniform'
```

```
Orthogonality = 'Orthogonal'

! Set Offsets for coordinates on this PE.

if (this_PE_X == 1) then
  Offset_PE_X   = zero
else
  Offset_PE_X   = SUM(NCells_X_Vector(1:this_PE_X-1)) * &
                  Lengths(1) / NCells_X_total
end if
if (NDimensions >= 2) then
  if (this_PE_Y == 1) then
    Offset_PE_Y = zero
  else
    Offset_PE_Y = SUM(NCells_Y_Vector(1:this_PE_Y-1)) * &
                  Lengths(2) / NCells_Y_total
  end if
end if
if (NDimensions == 3) then
  if (this_PE_Z == 1) then
    Offset_PE_Z = zero
  else
    Offset_PE_Z = SUM(NCells_Z_Vector(1:this_PE_Z-1)) * &
                  Lengths(3) / NCells_Z_total
  end if
end if

! Set Node Coordinates on this PE.

call Initialize (Coordinates_Nodes_PE, NDimensions, &
                 NNodes_Vector(this_PE), allocate_status(2))
node = 1
do pe_z = 1, NNodes_Z_Vector(this_PE_Z)
  do pe_y = 1, NNodes_Y_Vector(this_PE_Y)
    do pe_x = 1, NNodes_X_Vector(this_PE_X)
      Coordinates_Nodes_PE(1,node)   = &
          Offset_PE_X + (pe_x-1) * Lengths(1) / NCells_X_total
      if (NDimensions >= 2) then
        Coordinates_Nodes_PE(2,node) = &
          Offset_PE_Y + (pe_y-1) * Lengths(2) / NCells_Y_total
      end if
      if (NDimensions == 3) then
        Coordinates_Nodes_PE(3,node) = &
          Offset_PE_Z + (pe_z-1) * Lengths(3) / NCells_Z_total
      end if

      ! Increment node.
      node = node + 1

    end do
  end do
end do
VERIFY((node-1)==NNodes_Vector(this_PE),5)
```

```
! Initialize the Multi-Mesh object.

call Initialize_Base_Multi_Mesh (Mesh, NDimensions, Geometry, &
  Uniformity, Orthogonality, Structure, AMR, Shape, &
  NNodes_Vector, NCells_Vector, NFaces_Vector, &
  Coordinates_Nodes_PE, Nodes_of_Cells_PE, Mesh_Name, &
  allocate_status(3))

! Set Mesh%Cells_of_Cells_Index and Mesh%Flag_Faces_of_Cells.

call Initialize (Mesh%Cells_of_Cells_Index, Mesh%Cell_Structure, &
                 Mesh%Cell_Structure, &
                 Many_of_One_Array=Cells_of_Cells_PE, &
                 status=allocate_status(4))

call Initialize (Mesh%Flag_Faces_of_Cells, NCells_Vector(this_PE), &
                 NDimensions*2, allocate_status(5))
Mesh%Flag_Faces_of_Cells = Flag_Faces_of_Cells

! Set Uniform-mesh specific variables.

! Set physical dimensions of the mesh (Lengths).

call Initialize (Mesh%Lengths, NDimensions, allocate_status(6))
Mesh%Lengths = Lengths

! Set volume for all cells.

select case (NDimensions)
case (1)
  ! Suppressed Y, Z.
  Mesh%Volume_All_Cells = Lengths(1) / NCells_X_total
case (2)
  ! Suppressed Z.
  Mesh%Volume_All_Cells = Lengths(1) / NCells_X_total * &
                          Lengths(2) / NCells_Y_total
case (3)
  Mesh%Volume_All_Cells = Lengths(1) / NCells_X_total * &
                          Lengths(2) / NCells_Y_total * &
                          Lengths(3) / NCells_Z_total
end select

! Set area for all faces (3 types in 3-D).

call Initialize (Mesh%Area_All_Faces, NDimensions, allocate_status(7))
select case (NDimensions)
case (1)
  ! Suppressed Y, Z.
  Mesh%Area_All_Faces(1) = one
case (2)
  ! Suppressed Z.
  Mesh%Area_All_Faces(1) = Lengths(2) / NCells_Y_total
  Mesh%Area_All_Faces(2) = Lengths(1) / NCells_X_total
case (3)
```

```
      Mesh%Area_All_Faces(1) = Lengths(2) / NCells_Y_total * &
                               Lengths(3) / NCells_Z_total
      Mesh%Area_All_Faces(2) = Lengths(1) / NCells_X_total * &
                               Lengths(3) / NCells_Z_total
      Mesh%Area_All_Faces(3) = Lengths(1) / NCells_X_total * &
                               Lengths(2) / NCells_Y_total
    end select

    ! Finalize temporary variables.

    call Finalize (Cells_of_Cells_PE, allocate_status(8))
    call Finalize (Coordinates_Nodes_PE, allocate_status(9))
    call Finalize (Flag_Faces_of_Cells, allocate_status(10))
    call Finalize (NCells_Vector, allocate_status(11))
    call Finalize (NCells_X_Vector, allocate_status(12))
    call Finalize (NCells_Y_Vector, allocate_status(13))
    call Finalize (NCells_Z_Vector, allocate_status(14))
    call Finalize (NFaces_Vector, allocate_status(15))
    call Finalize (NNodes_Vector, allocate_status(16))
    call Finalize (NNodes_X_Vector, allocate_status(17))
    call Finalize (NNodes_Y_Vector, allocate_status(18))
    call Finalize (NNodes_Z_Vector, allocate_status(19))
    call Finalize (Nodes_of_Cells_PE, allocate_status(20))

    ! Consolidate and handle status.

    consolidated_status = allocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (allocate_status)

  end subroutine Initialize_Uniform_Multi_Mesh
```

I.1.3 Initialize_Orthogonal_Multi_Mesh Procedure

The main documentation of the Initialize_Orthogonal_Multi_Mesh Procedure in § 14.1.3 on page 174 contains additional explanation of this code listing.

```
  subroutine Initialize_Orthogonal_MMesh (Mesh, NDimensions, &
    Coordinates_Nodes_X, Coordinates_Nodes_Y, Coordinates_Nodes_Z, &
    Mesh_Name, status)

    ! Input variables.

    type(integer), intent(in) :: NDimensions      ! Number of Dimensions.
    type(real,1)           :: Coordinates_Nodes_X ! X-coordinates of the nodes.
    type(real,1), optional :: Coordinates_Nodes_Y ! Y-coordinates of the nodes.
```

```
type(real,1), optional :: Coordinates_Nodes_Z ! Z-coordinates of the nodes.
type(character,*), intent(in), optional :: Mesh_Name ! Mesh name.

! Output variables.

! Multi_Mesh to be initialized.
type(Multi_Mesh_type), intent(inout) :: Mesh
type(Status_type), intent(out), optional :: status  ! Exit status.

! Internal variables.

! Total number of cells in the X-, Y-, and Z-directions.
type(integer) :: NCells_X_total, NCells_Y_total, NCells_Z_total
type(character,name_length) :: Shape         ! Cell shape.
type(character,name_length) :: Geometry      ! Cell geometry (Cartesian).
type(character,name_length) :: Uniformity    ! Set to "Uniform".
type(character,name_length) :: Orthogonality ! Set to "Orthogonal".
type(character,name_length) :: Structure     ! Set to "Structured".
type(logical)               :: AMR           ! Set to false.
type(Status_type), dimension(20) :: allocate_status  ! Allocation Status.
type(Status_type) :: consolidated_status            ! Consolidated Status.
type(integer) :: NPEs_X ! Number of PEs in X.
type(integer) :: NPEs_Y ! Number of PEs in Y.
type(integer) :: NPEs_Z ! Number of PEs in Z.
type(real,1) :: Lengths      ! Physical extent of the
                             ! domain in each direction.

! Structure length vectors, which give numbers for all PEs.
type(integer,1) :: NCells_Vector   ! Number of cells.
type(integer,1) :: NCells_X_Vector ! Number of cells in the X direction.
type(integer,1) :: NCells_Y_Vector ! Number of cells in the Y direction.
type(integer,1) :: NCells_Z_Vector ! Number of cells in the Z direction.
type(integer,1) :: NFaces_Vector   ! Number of faces.
type(integer,1) :: NNodes_Vector   ! Number of nodes.
type(integer,1) :: NNodes_X_Vector ! Number of nodes in the X direction.
type(integer,1) :: NNodes_Y_Vector ! Number of nodes in the Y direction.
type(integer,1) :: NNodes_Z_Vector ! Number of nodes in the Z direction.

! Location of this_PE in the PE-mesh.
type(integer) :: this_PE_X, this_PE_Y, this_PE_Z
type(integer) :: node, pe_x, pe_y, pe_z     ! Loop parameters.

! Offsets - starting points for this PE.
type(integer) :: Offset_PE_X, Offset_PE_Y, Offset_PE_Z ! 1st node indices.

! Mesh coordinates and indices.

! The coordinates of the nodes on this PE.
type(real,2) :: Coordinates_Nodes_PE
! The nodes for the cells on this PE.
type(integer,2) :: Nodes_of_Cells_PE
! The cells for the cells (that is, across each face) on this PE.
type(integer,2) :: Cells_of_Cells_PE
! Face Flags for Structured Meshes.
```

```
type(integer,2) :: Flag_Faces_of_Cells

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

! Mesh type info is valid.
VERIFY(NDimensions .InInterval. (/1, 3/),5)

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)

! Set NCells_D_total and physical dimensions of the mesh - Lengths.

call Initialize (Lengths, NDimensions, allocate_status(1))
select case (NDimensions)
case (1)
  NCells_X_total = SIZE(Coordinates_Nodes_X) - 1
  NCells_Y_total = 1
  NCells_Z_total = 1
  Lengths(1) = Coordinates_Nodes_X(SIZE(Coordinates_Nodes_X)) &
             - Coordinates_Nodes_X(1)
case (2)
  NCells_X_total = SIZE(Coordinates_Nodes_X) - 1
  NCells_Y_total = SIZE(Coordinates_Nodes_Y) - 1
  NCells_Z_total = 1
  Lengths(1) = Coordinates_Nodes_X(SIZE(Coordinates_Nodes_X)) &
             - Coordinates_Nodes_X(1)
  Lengths(2) = Coordinates_Nodes_Y(SIZE(Coordinates_Nodes_Y)) &
             - Coordinates_Nodes_Y(1)
case (3)
  NCells_X_total = SIZE(Coordinates_Nodes_X) - 1
  NCells_Y_total = SIZE(Coordinates_Nodes_Y) - 1
  NCells_Z_total = SIZE(Coordinates_Nodes_Z) - 1
  Lengths(1) = Coordinates_Nodes_X(SIZE(Coordinates_Nodes_X)) &
             - Coordinates_Nodes_X(1)
  Lengths(2) = Coordinates_Nodes_Y(SIZE(Coordinates_Nodes_Y)) &
             - Coordinates_Nodes_Y(1)
  Lengths(3) = Coordinates_Nodes_Z(SIZE(Coordinates_Nodes_Z)) &
             - Coordinates_Nodes_Z(1)
end select

! Generate the connectivity.

call Gen_StructureMesh_Connectivity (NDimensions, Lengths, &
  NCells_X_total, NCells_Y_total, NCells_Z_total, Shape, Structure, AMR, &
  NPEs_X, NPEs_Y, NPEs_Z, this_PE_X, this_PE_Y, this_PE_Z, &
  NNodes_Vector, NCells_Vector, NFaces_Vector, &
  NCells_X_Vector, NCells_Y_Vector, NCells_Z_Vector, &
  NNodes_X_Vector, NNodes_Y_Vector, NNodes_Z_Vector, &
  Nodes_of_Cells_PE, Cells_of_Cells_PE, Flag_Faces_of_Cells, &
  allocate_status(2))
```

```
! Set mesh type info for an Orthogonal Mesh.

Geometry = 'Cartesian'
Uniformity = 'Nonuniform'
Orthogonality = 'Orthogonal'

! Set Offsets for coordinates on this PE. Note that this is
! the integer offset of the indices, not the physical distance
! offset as is used in the Uniform mesh initialization.

if (this_PE_X == 1) then
  Offset_PE_X   = 0
else
  Offset_PE_X   = SUM(NCells_X_Vector(1:this_PE_X-1))
end if
if (NDimensions >= 2) then
  if (this_PE_Y == 1) then
    Offset_PE_Y = 0
  else
    Offset_PE_Y = SUM(NCells_Y_Vector(1:this_PE_Y-1))
  end if
end if
if (NDimensions == 3) then
  if (this_PE_Z == 1) then
    Offset_PE_Z = 0
  else
    Offset_PE_Z = SUM(NCells_Z_Vector(1:this_PE_Z-1))
  end if
end if

! Set Node Coordinates on this PE.

call Initialize (Coordinates_Nodes_PE, NDimensions, &
                 NNodes_Vector(this_PE), allocate_status(3))
node = 1
do pe_z = 1, NNodes_Z_Vector(this_PE_Z)
  do pe_y = 1, NNodes_Y_Vector(this_PE_Y)
    do pe_x = 1, NNodes_X_Vector(this_PE_X)
      Coordinates_Nodes_PE(1,node)   = &
          Coordinates_Nodes_X(Offset_PE_X + pe_x)
      if (NDimensions >= 2) then
        Coordinates_Nodes_PE(2,node) = &
          Coordinates_Nodes_Y(Offset_PE_Y + pe_y)
      end if
      if (NDimensions == 3) then
        Coordinates_Nodes_PE(3,node) = &
          Coordinates_Nodes_Z(Offset_PE_Z + pe_z)
      end if

      ! Increment node.
      node = node + 1

    end do
```

```
    end do
  end do
  VERIFY((node-1)==NNodes_Vector(this_PE),5)

  ! Initialize the Multi-Mesh object.

  call Initialize_Base_Multi_Mesh (Mesh, NDimensions, Geometry, &
    Uniformity, Orthogonality, Structure, AMR, Shape, &
    NNodes_Vector, NCells_Vector, NFaces_Vector, &
    Coordinates_Nodes_PE, Nodes_of_Cells_PE, Mesh_Name, &
    allocate_status(4))

  ! Set physical dimensions of the mesh (Lengths).

  call Initialize (Mesh%Lengths, NDimensions, allocate_status(5))
  Mesh%Lengths = Lengths

  ! Set Mesh%Cells_of_Cells_Index and Mesh%Flag_Faces_of_Cells.

  call Initialize (Mesh%Cells_of_Cells_Index, Mesh%Cell_Structure, &
                   Mesh%Cell_Structure, &
                   Many_of_One_Array=Cells_of_Cells_PE, &
                   status=allocate_status(6))

  call Initialize (Mesh%Flag_Faces_of_Cells, NCells_Vector(this_PE), &
                   NDimensions*2, allocate_status(7))
  Mesh%Flag_Faces_of_Cells = Flag_Faces_of_Cells

  ! Finalize temporary variables.

  call Finalize (Cells_of_Cells_PE, allocate_status(8))
  call Finalize (Coordinates_Nodes_PE, allocate_status(9))
  call Finalize (Flag_Faces_of_Cells, allocate_status(10))
  call Finalize (NCells_Vector, allocate_status(11))
  call Finalize (NCells_X_Vector, allocate_status(12))
  call Finalize (NCells_Y_Vector, allocate_status(13))
  call Finalize (NCells_Z_Vector, allocate_status(14))
  call Finalize (NFaces_Vector, allocate_status(15))
  call Finalize (NNodes_Vector, allocate_status(16))
  call Finalize (NNodes_X_Vector, allocate_status(17))
  call Finalize (NNodes_Y_Vector, allocate_status(18))
  call Finalize (NNodes_Z_Vector, allocate_status(19))
  call Finalize (Nodes_of_Cells_PE, allocate_status(20))

  ! Consolidate and handle status.

  consolidated_status = allocate_status
  if (PRESENT(status)) then
    WARN_IF(Error(consolidated_status), 5)
    status = consolidated_status
  else
    VERIFY(Normal(consolidated_status), 5)
  end if
  call Finalize (consolidated_status)
```

```
    call Finalize (allocate_status)

  end subroutine Initialize_Orthogonal_MMesh

  subroutine Gen_StructureMesh_Connectivity (NDimensions, Lengths, &
      NCells_X_total, NCells_Y_total, NCells_Z_total, Shape, Structure, AMR, &
      NPEs_X, NPEs_Y, NPEs_Z, this_PE_X, this_PE_Y, this_PE_Z, &
      NNodes_Vector, NCells_Vector, NFaces_Vector, &
      NCells_X_Vector, NCells_Y_Vector, NCells_Z_Vector, &
      NNodes_X_Vector, NNodes_Y_Vector, NNodes_Z_Vector, &
      Nodes_of_Cells_PE, Cells_of_Cells_PE, Flag_Faces_of_Cells, status)

    ! Use associations.

    use Caesar_Mathematics_Module

    !~~~~~~~~~~~~~~~~~
    ! Input variables.
    !~~~~~~~~~~~~~~~~~

    ! Mesh type information.

    type(integer), intent(in) :: NDimensions    ! Number of Dimensions.
    type(real,1,np), intent(in) :: Lengths      ! Physical extent of the
                                                ! domain in each direction.
    ! Total number of cells in the X-, Y-, and Z-directions.
    type(integer), intent(in)              :: NCells_X_total
    type(integer), intent(inout), optional :: NCells_Y_total
    type(integer), intent(inout), optional :: NCells_Z_total

    !~~~~~~~~~~~~~~~~~~
    ! Output variables.
    !~~~~~~~~~~~~~~~~~~

    type(Status_type), intent(out), optional :: status  ! Exit status.
    type(character,name_length), intent(out) :: Shape     ! Cell shape.
    type(character,name_length), intent(out) :: Structure ! Set to Structured.
    type(logical), intent(out)               :: AMR       ! Set to false.

    ! Structure length vectors, which give numbers for all PEs.
    type(integer,1) :: NCells_Vector   ! Number of cells.
    type(integer,1) :: NFaces_Vector   ! Number of faces.
    type(integer,1) :: NNodes_Vector   ! Number of nodes.
    type(integer,1) :: NCells_X_Vector ! Number of cells in the X direction.
    type(integer,1) :: NCells_Y_Vector ! Number of cells in the Y direction.
    type(integer,1) :: NCells_Z_Vector ! Number of cells in the Z direction.
    type(integer,1) :: NNodes_X_Vector ! Number of nodes in the X direction.
    type(integer,1) :: NNodes_Y_Vector ! Number of nodes in the Y direction.
    type(integer,1) :: NNodes_Z_Vector ! Number of nodes in the Z direction.

    type(integer), intent(out) :: NPEs_X ! Number of PEs in X.
    type(integer), intent(out) :: NPEs_Y ! Number of PEs in Y.
    type(integer), intent(out) :: NPEs_Z ! Number of PEs in Z.
    ! Location of this_PE in the PE-mesh.
```

```
type(integer), intent(out) :: this_PE_X, this_PE_Y, this_PE_Z

! The nodes for the cells on this PE.
type(integer,2) :: Nodes_of_Cells_PE
! The cells for the cells (that is, across each face) on this PE.
type(integer,2) :: Cells_of_Cells_PE

! Face Flags for Structured Meshes. This is set to: 0 for internal,
! 1 for left (-x), 2 for right (+x), 3 for front (-y), 4 for back (+y),
! 5 for bottom (-z), 6 for top (+z).
type(integer,2) :: Flag_Faces_of_Cells

!~~~~~~~~~~~~~~~~~~~~
! Internal variables.
!~~~~~~~~~~~~~~~~~~~~

type(Status_type), dimension(18) :: allocate_status  ! Allocation Status.
type(Status_type) :: consolidated_status             ! Consolidated Status.
type(integer), dimension(32) :: NPE_Factors  ! Prime factors of NPEs.
type(integer) :: NNPE_Factors                ! Number of NPE Factors.

! Structure length vectors, which give numbers for all PEs.
type(integer) :: NCells_total      ! Total number of cells on all PEs.
type(integer) :: NFaces_total      ! Total number of faces on all PEs.
type(integer) :: NNodes_total      ! Total number of nodes on all PEs.

type(integer) :: i, j, k                             ! Loop parameters.
type(integer) :: cell, cell_x, cell_y, cell_z        ! Loop parameters.
type(integer) :: pe, pe_x, pe_y, pe_z                ! Loop parameters.
! Nodes_of_Cells evaluation point.
type(integer) :: node_cell_x, node_cell_y, node_cell_z, node_PE
type(integer) :: node_PE_X, node_PE_Y, node_PE_Z
! Cells_of_Cells evaluation point.
type(integer) :: other_cell_x, other_cell_y, other_cell_z, other_cell_PE
type(integer) :: other_cell_PE_X, other_cell_PE_Y, other_cell_PE_Z

! Offsets - starting points for this PE.

type(integer,1) :: Offset_Cells_Vector ! Global number of first cell.
type(integer,1) :: Offset_Faces_Vector ! Global number of first face.
type(integer,1) :: Offset_Nodes_Vector ! Global number of first node.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

! Mesh type info is valid.
VERIFY(NDimensions .InInterval. (/1, 3/),5)
VERIFY(SIZE(Lengths) == NDimensions,5)   ! Lengths is correct size.

! Allocations and initializations.

call Initialize (allocate_status)
call Initialize (consolidated_status)
```

```
! Initialize vectors.

call Initialize (NCells_Vector,       NPEs, allocate_status(1))
call Initialize (NFaces_Vector,       NPEs, allocate_status(2))
call Initialize (NNodes_Vector,       NPEs, allocate_status(3))
call Initialize (Offset_Cells_Vector, NPEs, allocate_status(4))
call Initialize (Offset_Faces_Vector, NPEs, allocate_status(5))
call Initialize (Offset_Nodes_Vector, NPEs, allocate_status(6))

! Set Shape, NCells_total based on NDimensions.

select case (NDimensions)
case (1)
  Shape = "Segmented    "
  NCells_total = NCells_X_total
  NCells_Y_total = 1
  NCells_Z_total = 1
  NFaces_total = NCells_total + 1
  NNodes_total = NCells_total + 1
case (2)
  Shape = "Quadrilateral"
  NCells_total = NCells_X_total * NCells_Y_total
  NCells_Z_total = 1
  NFaces_total = 2*NCells_total + NCells_X_total + NCells_Y_total
  NNodes_total = (NCells_X_total + 1) * (NCells_Y_total + 1)
case (3)
  Shape = "Hexahedral   "
  NCells_total = NCells_X_total * NCells_Y_total * NCells_Z_total
  NFaces_total = 3*NCells_total + NCells_X_total * NCells_Y_total  &
                          + NCells_Y_total * NCells_Z_total  &
                          + NCells_Z_total * NCells_X_total
  NNodes_total = (NCells_X_total + 1) * &
           (NCells_Y_total + 1) * &
           (NCells_Z_total + 1)
end select

! Set mesh type info for a Structured Mesh.

AMR = .false.
Structure = 'Structured'

! Determine parallel distributions: the number of PEs in each
! direction. The method used is to divide the problem between PEs so
! that the PE blocks look like a structured mesh with dimensions
! NPEs_X x NPEs_Y x NPEs_Z. Each block will have roughly the same number
! of cells, so load-balancing will be good (in 3D, each block will have
! roughly NCells_PE = (NCells_X_total/NPEs_X) * (NCells_Y_total/NPEs_Y) *
! (NCells_Z_total/NPEs_Z) ).
!
! To balance potential communication costs, the number of PEs in each
! direction are chosen to keep the physical dimensions of the blocks as
! equal as possible. The algorithm chosen is to prime factorize the
! total NPEs, then to iteratively add the largest remaining factor to the
```

```
! direction with the largest value of Length/NPEs until all the factors
! are used. This should make the blocks more equal (cubical), but it may
! not be the absolute optimal method. An additional requirement is that
! the NPEs in any direction must remain less than or equal to the number
! of cells in that direction.
!
! Better partitioning could be done if the overlying physics were known,
! however the mesh has no knowledge of the physics, and could indeed be
! used with different physics at different times.

if (NDimensions /= 1) then
  call Prime_Factors (NPEs, NNPE_Factors, NPE_Factors)
end if
select case (NDimensions)
case (1)
  NPEs_X = NPEs
  NPEs_Y = 1
  NPEs_Z = 1
  VERIFY(NPEs_X <= NCells_X_total,5)
case (2)
  NPEs_X = 1
  NPEs_Y = 1
  NPEs_Z = 1
  do while (NNPE_Factors > 0)

    ! Increase NPEs of direction with largest Length/NPEs, unless
    ! that would make NPEs > NCells in that direction.

    if (Lengths(1)/NPEs_X > Lengths(2)/NPEs_Y .and. &
        NPEs_X*NPE_Factors(NNPE_Factors) <= NCells_X_total) then
      NPEs_X = NPEs_X * NPE_Factors(NNPE_Factors)
      NNPE_Factors = NNPE_Factors - 1
    else if (NPEs_Y*NPE_Factors(NNPE_Factors) <= NCells_Y_total) then
      NPEs_Y = NPEs_Y * NPE_Factors(NNPE_Factors)
      NNPE_Factors = NNPE_Factors - 1

    ! If unsuccessful, then try to increase NPEs in other directions,
    ! retaining condition that NPEs must be less than or equal to
    ! NCells in that direction.

    else if (NPEs_X*NPE_Factors(NNPE_Factors) <= NCells_X_total) then
      NPEs_X = NPEs_X * NPE_Factors(NNPE_Factors)
      NNPE_Factors = NNPE_Factors - 1

    ! This condition is only reached if this mesh cannot be split up
    ! on the required number of PEs without leaving some PEs with
    ! no cells.

    else
      VERIFY(.false.,1)
    end if
  end do
case (3)
  NPEs_X = 1
```

```
    NPEs_Y = 1
    NPEs_Z = 1
    do while (NNPE_Factors > 0)

      ! Increase NPEs of direction with largest Length/NPEs, unless
      ! that would make NPEs > NCells in that direction.

      if (Lengths(1)/NPEs_X > Lengths(2)/NPEs_Y .and. &
          Lengths(1)/NPEs_X > Lengths(3)/NPEs_Z .and. &
          NPEs_X*NPE_Factors(NNPE_Factors) <= NCells_X_total) then
        NPEs_X = NPEs_X * NPE_Factors(NNPE_Factors)
        NNPE_Factors = NNPE_Factors - 1
      else if (Lengths(2)/NPEs_Y > Lengths(1)/NPEs_X .and. &
               Lengths(2)/NPEs_Y > Lengths(3)/NPEs_Z .and. &
               NPEs_Y*NPE_Factors(NNPE_Factors) <= NCells_Y_total) then
        NPEs_Y = NPEs_Y * NPE_Factors(NNPE_Factors)
        NNPE_Factors = NNPE_Factors - 1
      else if (NPEs_Z*NPE_Factors(NNPE_Factors) <= NCells_Z_total) then
        NPEs_Z = NPEs_Z * NPE_Factors(NNPE_Factors)
        NNPE_Factors = NNPE_Factors - 1

      ! If unsuccessful, then try to increase NPEs in other directions,
      ! retaining condition that NPEs must be less than or equal to
      ! NCells in that direction.

      else if (NPEs_X*NPE_Factors(NNPE_Factors) <= NCells_X_total) then
        NPEs_X = NPEs_X * NPE_Factors(NNPE_Factors)
        NNPE_Factors = NNPE_Factors - 1
      else if (NPEs_Y*NPE_Factors(NNPE_Factors) <= NCells_Y_total) then
        NPEs_Y = NPEs_Y * NPE_Factors(NNPE_Factors)
        NNPE_Factors = NNPE_Factors - 1

      ! This condition is only reached if this mesh cannot be split up
      ! on the required number of PEs without leaving some PEs with
      ! no cells.

      else
        VERIFY(.false.,1)
      end if
    end do
  end select

  ! Initialize NCells_D_Vectors, NNodes_D_Vectors.

  call Initialize (NCells_X_Vector, NPEs_X, allocate_status(7))
  call Initialize (NCells_Y_Vector, NPEs_Y, allocate_status(8))
  call Initialize (NCells_Z_Vector, NPEs_Z, allocate_status(9))
  call Initialize (NNodes_X_Vector, NPEs_X, allocate_status(10))
  call Initialize (NNodes_Y_Vector, NPEs_Y, allocate_status(11))
  call Initialize (NNodes_Z_Vector, NPEs_Z, allocate_status(12))

  ! Set NCells_D_Vectors (divide evenly among the PEs).

  call Generate_Even_Distribution (NCells_X_Vector, NCells_X_total)
```

```
call Generate_Even_Distribution (NCells_Y_Vector, NCells_Y_total)
call Generate_Even_Distribution (NCells_Z_Vector, NCells_Z_total)

! Set NNodes_D_Vectors -- extra node on last PE in each direction.

NNodes_X_Vector = NCells_X_Vector
NNodes_Y_Vector = NCells_Y_Vector
NNodes_Z_Vector = NCells_Z_Vector
NNodes_X_Vector(NPEs_X) = NNodes_X_Vector(NPEs_X) + 1
if (NDimensions > 1) then
  NNodes_Y_Vector(NPEs_Y) = NNodes_Y_Vector(NPEs_Y) + 1
  if (NDimensions > 2) then
    NNodes_Z_Vector(NPEs_Z) = NNodes_Z_Vector(NPEs_Z) + 1
  end if
end if

! Set NCells_Vector, NFaces_Vector, and NNodes_Vector.

pe = 1
do pe_z = 1, NPEs_Z
  do pe_y = 1, NPEs_Y
    do pe_x = 1, NPEs_X

      ! Set NCells_Vector.

      NCells_Vector(pe) = NCells_X_Vector(pe_x) * &
                          NCells_Y_Vector(pe_y) * &
                          NCells_Z_Vector(pe_z)

      ! Set NFaces_Vector.

      NFaces_Vector(pe) = NDimensions * NCells_Vector(pe)
      ! Add extra YZ faces for pe_x = NPEs_X plane.
      if (pe_x == NPEs_X) then
        NFaces_Vector(pe) = NFaces_Vector(pe) + &
                            NCells_Y_Vector(pe_y) * &
                            NCells_Z_Vector(pe_z)
      end if
      ! Add extra XZ faces for pe_y = NPEs_Y plane in 2-D and 3-D.
      if (pe_y == NPEs_Y .and. NDimensions /= 1) then
        NFaces_Vector(pe) = NFaces_Vector(pe) + &
                            NCells_X_Vector(pe_x) * &
                            NCells_Z_Vector(pe_z)
      end if
      ! Add extra XY faces for pe_z = NPEs_Z plane in 3-D.
      if (pe_z == NPEs_Z .and. NDimensions == 3) then
        NFaces_Vector(pe) = NFaces_Vector(pe) + &
                            NCells_X_Vector(pe_x) * &
                            NCells_Y_Vector(pe_y)
      end if

      ! Set NNodes_Vector.

      NNodes_Vector(pe) = NNodes_X_Vector(pe_x) * &
```

```
                              NNodes_Y_Vector(pe_y) * &
                              NNodes_Z_Vector(pe_z)

          ! Capture PE info.

          if (pe == this_PE) then
            this_PE_X = pe_x
            this_PE_Y = pe_y
            this_PE_Z = pe_z
          end if

          ! Increment pe.
          pe = pe + 1

        end do
      end do
    end do
    VERIFY((pe-1)==NPEs,8)
    VERIFY(SUM(NCells_Vector)==NCells_total,8)
    VERIFY(SUM(NFaces_Vector)==NFaces_total,8)
    VERIFY(SUM(NNodes_Vector)==NNodes_total,8)

    ! Set Offset Vectors.

    Offset_Cells_Vector(1) = 1
    Offset_Nodes_Vector(1) = 1
    Offset_Faces_Vector(1) = 1
    if (NPEs > 1) then
      pe = 1
      do pe_z = 1, NPEs_Z
        do pe_y = 1, NPEs_Y
          do pe_x = 1, NPEs_X

            ! Skip first pe -- it's already done.
            if (pe/=1) then

              ! Set Offset_Cells_Vector.

              Offset_Cells_Vector(pe) = Offset_Cells_Vector(pe-1) + &
                                        NCells_Vector(pe-1)

              ! Set Offset_Nodes_Vector.

              Offset_Nodes_Vector(pe) = Offset_Nodes_Vector(pe-1) + &
                                        NNodes_Vector(pe-1)

              ! Set Offset_Faces_Vector.

              Offset_Faces_Vector(pe) = Offset_Faces_Vector(pe-1) + &
                                        NFaces_Vector(pe-1)
            end if

            ! Increment pe.
            pe = pe + 1
```

```
      end do
    end do
  end do
end if

! Set Nodes_of_Cells_PE.

call Initialize (Nodes_of_Cells_PE, NCells_Vector(this_PE), &
                 2**NDimensions, allocate_status(13))
cell = 1
do cell_z = 1, NCells_Z_Vector(this_PE_Z)
  do cell_y = 1, NCells_Y_Vector(this_PE_Y)
    do cell_x = 1, NCells_X_Vector(this_PE_X)
      ! These next three loops loop over the nodes in a cell.
      do k = 0, 1
        do j = 0, 1
          do i = 0, 1

            ! Determine which PE the node is on, and set
            ! node_cell_D and node_PE to reflect this.

            node_cell_x = cell_x + i
            node_cell_y = cell_y + j
            node_cell_z = cell_z + k
            node_PE = this_PE
            node_PE_X = this_PE_X
            node_PE_Y = this_PE_Y
            node_PE_Z = this_PE_Z
            if (node_cell_x > NNodes_X_Vector(this_PE_X)) then
              node_cell_x = 1
              node_PE = node_PE + 1
              node_PE_X = node_PE_X + 1
            end if
            if (node_cell_y > NNodes_Y_Vector(this_PE_Y)) then
              node_cell_y = 1
              node_PE = node_PE + NPEs_X
              node_PE_Y = node_PE_Y + 1
            end if
            if (node_cell_z > NNodes_Z_Vector(this_PE_Z)) then
              node_cell_z = 1
              node_PE = node_PE + NPEs_X * NPEs_Y
              node_PE_Z = node_PE_Z + 1
            end if

            ! Use formula to set node number for this node within the
            ! cell, now that the PE that the node is on is known. Note
            ! that the generalized multi-dimensional node numbering
            ! (within a cell) is used.

            Nodes_of_Cells_PE(cell,i + 2*j + 4*k + 1) = &
              Offset_Nodes_Vector(node_PE) + &
              (node_cell_x - 1) + &
              (node_cell_y - 1) * NNodes_X_Vector(node_PE_X) + &
```

```
                (node_cell_z - 1) * NNodes_X_Vector(node_PE_X) * &
                                    NNodes_Y_Vector(node_PE_Y)

            end do
            if (NDimensions < 2) exit
          end do
          if (NDimensions < 3) exit
        end do

        ! Increment cell.
        cell = cell + 1

      end do
    end do
  end do

  ! Set Cells_of_Cells_PE and Flag_Faces_of_Cells.

  call Initialize (Cells_of_Cells_PE, NCells_Vector(this_PE), &
                   NDimensions*2, allocate_status(14))
  call Initialize (Flag_Faces_of_Cells, NCells_Vector(this_PE), &
                   NDimensions*2, allocate_status(15))
  cell = 0
  do cell_z = 1, NCells_Z_Vector(this_PE_Z)
    do cell_y = 1, NCells_Y_Vector(this_PE_Y)
      do cell_x = 1, NCells_X_Vector(this_PE_X)

        ! Increment cell.

        cell = cell + 1

        ! Loop over X Faces.

        do i = -1, 1, 2

          ! Determine which PE the "other" cell is on, and set
          ! other_cell_D and other_cell_PE to reflect this. Notice
          ! the loop-around which is done to allow periodic boundary
          ! conditions.

          other_cell_x = cell_x + i
          other_cell_y = cell_y
          other_cell_z = cell_z
          other_cell_PE   = this_PE
          other_cell_PE_X = this_PE_X
          other_cell_PE_Y = this_PE_Y
          other_cell_PE_Z = this_PE_Z
          if (other_cell_x > NCells_X_Vector(this_PE_X)) then
            other_cell_PE   = other_cell_PE   + 1
            other_cell_PE_X = other_cell_PE_X + 1
            if (other_cell_PE_X > NPEs_X) then
              other_cell_PE_X = other_cell_PE_X - NPEs_X ! Should = 1.
              other_cell_PE   = other_cell_PE   - NPEs_X
              Flag_Faces_of_Cells(cell,2) = 2
```

```
      end if
      other_cell_x = 1
    end if
    if (other_cell_x < 1) then
      other_cell_PE   = other_cell_PE   - 1
      other_cell_PE_X = other_cell_PE_X - 1
      if (other_cell_PE_X < 1) then
        other_cell_PE_X = other_cell_PE_X + NPEs_X ! Should = NPEs_X.
        other_cell_PE   = other_cell_PE   + NPEs_X
        Flag_Faces_of_Cells(cell,1) = 1
      end if
      other_cell_x = NCells_X_Vector(other_cell_PE_X)
    end if

    ! Use formula to set cell number for the "other" cell, now
    ! that the PE that the other cell is on is known. Note
    ! that the generalized multi-dimensional node numbering
    ! (within a cell) is used. This sets values for faces 1 and 2.

    Cells_of_Cells_PE(cell,(i + 3)/2) = &
      Offset_Cells_Vector(other_cell_PE) + &
      (other_cell_x - 1) + &
      (other_cell_y - 1) * NCells_X_Vector(other_cell_PE_X) + &
      (other_cell_z - 1) * NCells_X_Vector(other_cell_PE_X) * &
                           NCells_Y_Vector(other_cell_PE_Y)

  end do
  if (NDimensions < 2) cycle

  ! Loop over Y Faces.

  do j = -1, 1, 2

    ! Determine which PE the "other" cell is on, and set
    ! other_cell_D and other_cell_PE to reflect this. Notice
    ! the loop-around which is done to allow periodic boundary
    ! conditions.

    other_cell_x = cell_x
    other_cell_y = cell_y + j
    other_cell_z = cell_z
    other_cell_PE   = this_PE
    other_cell_PE_X = this_PE_X
    other_cell_PE_Y = this_PE_Y
    other_cell_PE_Z = this_PE_Z
    if (other_cell_y > NCells_Y_Vector(this_PE_Y)) then
      other_cell_PE   = other_cell_PE   + NPEs_X
      other_cell_PE_Y = other_cell_PE_Y + 1
      if (other_cell_PE_Y > NPEs_Y) then
        other_cell_PE_Y = other_cell_PE_Y - NPEs_Y ! Should = 1.
        other_cell_PE   = other_cell_PE   - NPEs_Y * NPEs_X
        Flag_Faces_of_Cells(cell,4) = 4
      end if
      other_cell_y = 1
```

```
      end if
      if (other_cell_y < 1) then
        other_cell_PE   = other_cell_PE   - NPEs_X
        other_cell_PE_Y = other_cell_PE_Y - 1
        if (other_cell_PE_Y < 1) then
          other_cell_PE_Y = other_cell_PE_Y + NPEs_Y ! Should = NPEs_Y.
          other_cell_PE   = other_cell_PE   + NPEs_Y * NPEs_X
          Flag_Faces_of_Cells(cell,3) = 3
        end if
        other_cell_y = NCells_Y_Vector(other_cell_PE_Y)
      end if

      ! Use formula to set cell number for the "other" cell, now
      ! that the PE that the other cell is on is known. Note
      ! that the generalized multi-dimensional node numbering
      ! (within a cell) is used. This sets values for faces 3 and 4.

      Cells_of_Cells_PE(cell,(j + 7)/2) = &
        Offset_Cells_Vector(other_cell_PE) + &
        (other_cell_x - 1) + &
        (other_cell_y - 1) * NCells_X_Vector(other_cell_PE_X) + &
        (other_cell_z - 1) * NCells_X_Vector(other_cell_PE_X) * &
                             NCells_Y_Vector(other_cell_PE_Y)

    end do
    if (NDimensions < 3) cycle

    ! Loop over Z Faces.

    do k = -1, 1, 2

      ! Determine which PE the "other" cell is on, and set
      ! other_cell_D and other_cell_PE to reflect this. Notice
      ! the loop-around which is done to allow periodic boundary
      ! conditions.

      other_cell_x = cell_x
      other_cell_y = cell_y
      other_cell_z = cell_z + k
      other_cell_PE   = this_PE
      other_cell_PE_X = this_PE_X
      other_cell_PE_Y = this_PE_Y
      other_cell_PE_Z = this_PE_Z
      if (other_cell_z > NCells_Z_Vector(this_PE_Z)) then
        other_cell_PE   = other_cell_PE   + NPEs_X * NPEs_Y
        other_cell_PE_Z = other_cell_PE_Z + 1
        if (other_cell_PE_Z > NPEs_Z) then
          other_cell_PE_Z = other_cell_PE_Z - NPEs_Z ! Should = 1.
          other_cell_PE   = other_cell_PE   - NPEs_Z * NPEs_X * NPEs_Y
          Flag_Faces_of_Cells(cell,6) = 6
        end if
        other_cell_z = 1
      end if
      if (other_cell_z < 1) then
```

```
            other_cell_PE   = other_cell_PE   - NPEs_X * NPEs_Y
            other_cell_PE_Z = other_cell_PE_Z - 1
            if (other_cell_PE_Z < 1) then
              other_cell_PE_Z = other_cell_PE_Z + NPEs_Z ! Should = NPEs_Z.
              other_cell_PE   = other_cell_PE   + NPEs_Z * NPEs_X * NPEs_Y
              Flag_Faces_of_Cells(cell,5) = 5
            end if
            other_cell_z = NCells_Z_Vector(other_cell_PE_Z)
          end if

          ! Use formula to set cell number for the "other" cell, now
          ! that the PE that the other cell is on is known. Note
          ! that the generalized multi-dimensional node numbering
          ! (within a cell) is used. This sets values for faces 5 and 6.

          Cells_of_Cells_PE(cell,(k + 11)/2) = &
            Offset_Cells_Vector(other_cell_PE) + &
            (other_cell_x - 1) + &
            (other_cell_y - 1) * NCells_X_Vector(other_cell_PE_X) + &
            (other_cell_z - 1) * NCells_X_Vector(other_cell_PE_X) * &
                                 NCells_Y_Vector(other_cell_PE_Y)

        end do

      end do
    end do
  end do

  ! Finalize temporary variables.

  call Finalize (Offset_Cells_Vector, allocate_status(16))
  call Finalize (Offset_Faces_Vector, allocate_status(17))
  call Finalize (Offset_Nodes_Vector, allocate_status(18))

  ! Consolidate and handle status.

  consolidated_status = allocate_status
  if (PRESENT(status)) then
    WARN_IF(Error(consolidated_status), 5)
    status = consolidated_status
  else
    VERIFY(Normal(consolidated_status), 5)
  end if
  call Finalize (consolidated_status)
  call Finalize (allocate_status)

  return
end subroutine Gen_StructureMesh_Connectivity
```

I.1.4 Finalize_Multi_Mesh Procedure

The main documentation of the Finalize_Multi_Mesh Procedure in § 14.1.4 on page 174 contains additional explanation of this code listing.

```
  subroutine Finalize_Multi_Mesh (Mesh, status)

    ! Use associations.

    use Caesar_Flags_Module, only: uninitialized_flag

    ! Input/Output variable.

    ! Multi_Mesh to be finalized.
    type(Multi_Mesh_type), intent(inout) :: Mesh

    ! Output variable.

    type(Status_type), intent(out), optional :: status   ! Exit status.

    ! Internal variables.

    type(Status_type), dimension(4) :: deallocate_status ! Deallocation Status.
    type(Status_type) :: consolidated_status             ! Consolidated Status.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),7) ! Mesh is valid.

    ! Unset initialization flag.

    Mesh%Initialized = uninitialized_flag

    ! Set deallocation status.

    call Initialize (deallocate_status)
    call Initialize (consolidated_status)

    ! ### This procedure could use some work. All mesh variables
    ! ### are not finalized here yet.

    ! Finalize internals.

    !NULLIFY(Mesh%One_Structure)
    !NULLIFY(Mesh%Many_Structure)
    !NULLIFY(Mesh%Many_of_One_Index)

    !select case (Mesh%A_Dimensionality)
    !case (1)
    !  call Finalize (Mesh%Values1, deallocate_status(1))
    !case (2)
```

```
    !  call Finalize (Mesh%Values2, deallocate_status(1))
    !case (3)
    !  call Finalize (Mesh%Values3, deallocate_status(1))
    !case (4)
    !  call Finalize (Mesh%Values4, deallocate_status(1))
    !case (5)
    !  call Finalize (Mesh%Values5, deallocate_status(1))
    !case (-1)
    !  call Finalize (Mesh%ValuesRR, deallocate_status(1))
    !end select
    !call Finalize (Mesh%A_Dimensionality, deallocate_status(2))
    !call Finalize (Mesh%Dimensionality,   deallocate_status(3))
    call Finalize (Mesh%NDimensions,      deallocate_status(1))
    call Finalize (Mesh%Name,             deallocate_status(2))
    call Finalize (Mesh%Version,          deallocate_status(3))
    if (Mesh%Orthogonality == 'Orthogonal') then
      call Finalize (Mesh%Lengths,        deallocate_status(4))
    end if

    ! Process status variables.

    consolidated_status = deallocate_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (deallocate_status)

    ! Verify guarantees.

    VERIFY(.not.Valid_State(Mesh),7) ! Mesh is not valid.

    return
  end subroutine Finalize_Multi_Mesh
```

I.1.5 Valid_State_Multi_Mesh Procedure

The main documentation of the Valid_State_Multi_Mesh Procedure in § 14.1.5 on page 175 contains additional explanation of this code listing.

```
  function Valid_State_Multi_Mesh (Mesh) result(Valid)

    ! Input variables.

    ! Variable to be checked.
    type(Multi_Mesh_type), intent(in) :: Mesh

    ! Output variables.
```

```
type(logical) :: Valid       ! Logical state.

! Internal variables.

!type(integer) :: NSlice      ! Number of Values in a "slice" of the Mesh.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Start out true.

Valid = .true.

! Check for association of pointered internals.

!Valid = Valid .and. ASSOCIATED(Mesh%One_Structure)
!Valid = Valid .and. ASSOCIATED(Mesh%Many_Structure)
!Valid = Valid .and. ASSOCIATED(Mesh%Many_of_One_Index)
!if (.not.Valid) return

! Check for validity of internals.

!VERIFY(NDimensions .eq. 1 .eqv. Shape .eq. "Segmented",5)
!etc.

Valid = Valid .and. Initialized(Mesh)
!Valid = Valid .and. Valid_State(Mesh%A_Dimensionality)
!Valid = Valid .and. Valid_State(Mesh%Dimensionality)
Valid = Valid .and. Valid_State(Mesh%NDimensions)
!Valid = Valid .and. Valid_State(Mesh%Many_Structure)
!Valid = Valid .and. Valid_State(Mesh%Many_of_One_Index)
Valid = Valid .and. Valid_State(Mesh%Name)
!Valid = Valid .and. Valid_State(Mesh%One_Structure)
!select case (Mesh%A_Dimensionality)
!case (1)
!  Valid = Valid .and. Valid_State(Mesh%Values1)
!case (2)
!  Valid = Valid .and. Valid_State(Mesh%Values2)
!case (3)
!  Valid = Valid .and. Valid_State(Mesh%Values3)
!case (4)
!  Valid = Valid .and. Valid_State(Mesh%Values4)
!case (5)
!  Valid = Valid .and. Valid_State(Mesh%Values5)
!case (-1)
!  Valid = Valid .and. Valid_State(Mesh%ValuesRR)
!end select
Valid = Valid .and. Valid_State(Mesh%Version)
if (.not.Valid) return

! Checks on the validity of Mesh.

!Valid = Valid .and. Mesh%A_Dimensionality == &
!        Mesh%Dimensionality + Mesh%Many_of_One_Index%Dimensionality - 1
```

```
    return
  end function Valid_State_Multi_Mesh
```

I.1.6 Initialized_Multi_Mesh Procedure

The main documentation of the Initialized_Multi_Mesh Procedure in § 14.1.6 on page 175 contains additional explanation of this code listing.

```
  function Initialized_Multi_Mesh (Mesh) result(Initialized)

    ! Use associations.

    use Caesar_Flags_Module, only: initialized_flag

    ! Input variable.

    ! Multi_Mesh to be checked.
    type(Multi_Mesh_type), intent(in) :: Mesh

    ! Output variable.

    type(logical) :: Initialized          ! Initialized condition boolean.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements - none.

    ! Set initialized boolean.

    Initialized = Mesh%Initialized == initialized_flag

    ! Verify guarantees - none.

    return
  end function Initialized_Multi_Mesh
```

I.1.7 Dump_CGNS_Multi_Mesh Procedure

The main documentation of the Dump_CGNS_Multi_Mesh Procedure in § 14.1.7 on page 176 contains additional explanation of this code listing.

```
  ifdef([USE_CGNSLIB],[
    subroutine Dump_CGNS_Multi_Mesh (Mesh, Filename, status)

      ! Input variables.

      type(Multi_Mesh_type), intent(in) :: Mesh       ! Mesh to be output.
      type(character,*), intent(in) :: Filename       ! Output filename.
```

```
! Output variable.

type(Status_type), optional :: status           ! Consolidated Status.

! Internal variables.

type(integer) :: CGNS_Base_Index                ! CGNS base index number.
type(integer) :: CGNS_File_Index                ! CGNS file index number.
type(integer) :: CGNS_Zone_Index                ! CGNS zone index number.
type(integer) :: CGNS_Status                    ! CGNS status.
type(Status_type) :: consolidated_status        ! Consolidated Status.
type(character,10), dimension(3) :: Coordinate_Name ! Coordinate names.
type(Status_type), dimension(13) :: dump_status ! Status vector.
type(integer) :: Element_Type                   ! Element type number.
type(integer) :: Section_Number                 ! Section number.

! Included CGNS setup file.

include (../../../external/CGNSLib/cgnslib_f.h)

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Mesh),5)      ! Mesh is valid.

! Allocations and initializations.

call Initialize (dump_status)
call Initialize (consolidated_status)

! Open CGNS file for writing, get CGNS_File_Index.

if (this_is_IO_PE) then
  call CG_Open_f (Filename, MODE_WRITE, CGNS_File_Index, CGNS_Status)
end if
if (CGNS_Status == ERROR) dump_status(1) = 'CGNS Error'

! Write CGNS Base information (dimensionality info), get CGNS_Base_Index.

if (this_is_IO_PE) then
  call CG_Base_Write_f (CGNS_File_Index, 'Caesar Base', &
                        Mesh%NDimensions, Mesh%NDimensions, &
                        CGNS_Base_Index, CGNS_Status)
end if
if (CGNS_Status == ERROR) dump_status(2) = 'CGNS Error'

! ### Add if-check on "structured" here eventually.

! Write CGNS Zone information (mesh dimension info), get CGNS_Zone_Index.

if (this_is_IO_PE) then
  call CG_Zone_Write_f (CGNS_File_Index, CGNS_Base_Index, Name_Name, &
                        (/ Mesh%NNodes_total, Mesh%NCells_total, 0 /), &
```

```
                        Unstructured, CGNS_Zone_Index, CGNS_Status)
end if
if (CGNS_Status == ERROR) dump_status(3) = 'CGNS Error'

! Define Coordinate Names.

if (Mesh%Geometry == 'Cartesian') then
  Coordinate_Name(1) = 'CoordinateX'
  Coordinate_Name(2) = 'CoordinateY'
  Coordinate_Name(3) = 'CoordinateZ'
else if (Mesh%Geometry == 'Cylindrical') then
  Coordinate_Name(1) = 'CoordinateR'
  Coordinate_Name(2) = 'CoordinateZ'
else if (Mesh%Geometry == 'Spherical') then
  Coordinate_Name(1) = 'CoordinateR'
end if

! Define DataType parameter.

ifdef([SINGLE],[
  define([CGNS_DataType],[RealSingle])
],[
  define([CGNS_DataType],[RealDouble])
])

! Write CGNS Zone coordinates (node coordinates),
! after assembling on the IO PE.

call Initialize (Coordinates_Nodes_AV, Mesh%Node_Structure, &
                 2, 'Coordinates of Nodes', dump_status(4), &
                 Mesh%NDimensions)
call Initialize (Coordinates_Nodes_BNV, Mesh%NDimensions, &
                 Mesh%NNodes_total, dump_status(5))
Coordinates_Nodes_AV = Mesh%Coordinates_Nodes_DV
Coordinates_Nodes_BNV = Coordinates_Nodes_AV
if (this_is_IO_PE) then
  do dim = 1, Mesh%NDimensions
    call CG_Coord_Write_f (CGNS_File_Index, CGNS_Base_Index, &
                           CGNS_Zone_Index, CGNS_DataType, &
                           Coordinate_Name(dim), &
                           Coordinates_Nodes_BNV(dim,:), &
                           dim, CGNS_Status)
  end do
end if
call Finalize (Coordinates_Nodes_BNV, dump_status(6))
call Finalize (Coordinates_Nodes_AV, dump_status(7))
if (CGNS_Status == ERROR) dump_status(8) = 'CGNS Error'

! Write CGNS Elements Connectivity (mesh connectivity),
! after assembling on the IO PE.

if (this_is_IO_PE) then
  Index_Size = Mesh%NCells_total
else
```

```
    Index_Size = 0
  end if
  call Initialize (Nodes_of_Cells_Index_Val_PE, Mesh%NCells_PE, &
                   Mesh%Nodes_per_Cell, dump_status(9))
  call Initialize (Nodes_of_Cells_Index_Val_Total, Index_Size, &
                   Mesh%Nodes_per_Cell, dump_status(10))
  Nodes_of_Cells_Index_Val_PE = Mesh%Nodes_of_Cells_Index
  do node = 1, Mesh%Nodes_per_Cell
    call Assemble (Nodes_of_Cells_Index_Val_Total(:,node), &
                   Nodes_of_Cells_Index_Val_PE(:,node))
  end do
  call Initialize (CGNS_Elements, Mesh%Nodes_per_Cell, Index_Size, &
                   dump_status(11))
  do cell = 1, Mesh%NCells

    select case (Mesh%Shape)  ! Select on Mesh%Shape.

    case ('Segmented')
      Element_Type = BAR_2
      CGNS_Elements(:,cell) = Nodes_of_Cells_Index_Val_Total(cell,:)
    case ('Triangular')
      Element_Type = TRI_3
      ! Note: it is unclear what order the nodes are in for triangles
      ! in CGNS format. It is assumed that the order is counterclockwise.
      CGNS_Elements(:,cell) = Nodes_of_Cells_Index_Val_Total(cell,:)
    case ('Quadrilateral')
      Element_Type = QUAD_4
      ! Note: it is unclear what order the nodes are in for quadrilaterals
      ! in CGNS format. It is assumed that the order is counterclockwise.
      CGNS_Elements(1,cell) = Nodes_of_Cells_Index_Val_Total(cell,1)
      CGNS_Elements(2,cell) = Nodes_of_Cells_Index_Val_Total(cell,2)
      CGNS_Elements(3,cell) = Nodes_of_Cells_Index_Val_Total(cell,4)
      CGNS_Elements(4,cell) = Nodes_of_Cells_Index_Val_Total(cell,3)
    case ('Polygonal')
      VERIFY(.false.,0)   ! Not implemented yet.
    case ('Tetrahedral')
      Element_Type = TETRA_4
      CGNS_Elements(:,cell) = Nodes_of_Cells_Index_Val_Total(cell,:)
    case ('Hexahedral')
      Element_Type = HEXA_8
      CGNS_Elements(1,cell) = Nodes_of_Cells_Index_Val_Total(cell,1)
      CGNS_Elements(2,cell) = Nodes_of_Cells_Index_Val_Total(cell,2)
      CGNS_Elements(3,cell) = Nodes_of_Cells_Index_Val_Total(cell,4)
      CGNS_Elements(4,cell) = Nodes_of_Cells_Index_Val_Total(cell,3)
      CGNS_Elements(5,cell) = Nodes_of_Cells_Index_Val_Total(cell,5)
      CGNS_Elements(6,cell) = Nodes_of_Cells_Index_Val_Total(cell,6)
      CGNS_Elements(7,cell) = Nodes_of_Cells_Index_Val_Total(cell,8)
      CGNS_Elements(8,cell) = Nodes_of_Cells_Index_Val_Total(cell,7)
    case ('Polyhedral')
      VERIFY(.false.,0)   ! Not implemented yet.
    end select
  end do
  if (this_is_IO_PE) then
    call CG_Section_Write_f (CGNS_File_Index, CGNS_Base_Index, &
```

```
                                CGNS_Zone_Index, 'VolumeElements', &
                                Element_Type, 1, Mesh%NCells, 0, &
                                CGNS_Elements, S, CGNS_Status)
      end if
      call Finalize (Nodes_of_Cells_Index_Val_PE, dump_status(12))
      if (CGNS_Status == ERROR) dump_status(12) = 'CGNS Error'

      ! Close CGNS file.

      if (this_is_IO_PE) then
        call CG_Close_f (CGNS_File_Index, CGNS_Status)
      end if
      if (CGNS_Status == ERROR) dump_status(13) = 'CGNS Error'

      ! Consolidate and handle status.

      consolidated_status = dump_status
      if (PRESENT(status)) then
        WARN_IF(Error(consolidated_status), 5)
        status = consolidated_status
      else
        VERIFY(Normal(consolidated_status), 5)
      end if
      call Finalize (consolidated_status)
      call Finalize (dump_status)

      ! Verify guarantees - none.

      return
    end subroutine Dump_CGNS_Multi_Mesh
  ])
```

I.1.8 Dump_GMV_Multi_Mesh Procedure

The main documentation of the Dump_GMV_Multi_Mesh Procedure in § 14.1.8 on page 176 contains additional explanation of this code listing.

```
  ! Set number of Mathematic Vectors and Distributed Vectors allowed
  ! in the call line. This can be changed here and it will change all
  ! necessary code to match. This makes use of the replicate m4 macros.
  ! Setting this to a number greater than 6 triggers errors on compilers
  ! that limit lines to 132 characters (like Absoft).

  include(replicate.m4)
  define(REP_NUMBER, 6)

  subroutine Dump_GMV_Multi_Mesh (Filename, Mesh &
    REP_ARGS([Variable[]i[]_MV]) &
    REP_ARGS([Variable[]i[]_DV]) &
    , status)

    ! Input variables.
```

```
type(Multi_Mesh_type), intent(in) :: Mesh       ! Mesh to be output.
type(character,*), intent(in) :: Filename       ! Output filename.
! Output Mathematic_Vector variables.
REP_DECLARE([type(Mathematic_Vector_type), optional], [Variable[]i[]_MV])
! Output Distributed_Vector variables.
REP_DECLARE([type(Distributed_Vector_type), optional], [Variable[]i[]_DV])

! Output variable.

type(Status_type), optional :: status           ! Consolidated Status.

! Internal variables.

type(integer) :: GMV_Status                     ! GMV file open status.
type(integer) :: unit                           ! GMV output unit.
type(integer) :: cell, node                     ! Loop parameters.

type(Assembled_Vector_type) :: Coordinates_Nodes_AV ! Node Coordinates AV.
type(real,2) :: Coordinates_Nodes_BNV             ! Node Coordinates BNV.
! Values from the Nodes_of_Cells_Index for this PE.
type(integer,2) :: Nodes_of_Cells_Index_Val_PE
! Values from the Nodes_of_Cells_Index for all PEs.
type(integer,2) :: Nodes_of_Cells_Index_Val_Total
type(integer) :: Index_Size
! Status vector.
type(Status_type), dimension(10+2*REP_NUMBER) :: dump_status
type(Status_type) :: consolidated_status           ! Consolidated Status.
ifelse(COMPILER, Lahey, [
  type(character,256) :: Error_Message ! String for compiler error message.
])

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Mesh),5)      ! Mesh is valid.
! Variable#_(DV|MV) is valid.
define([VERIFY_VARIABLE],[
  if (PRESENT(Variable$1_$2)) then
    VERIFY(Valid_State(Variable$1_$2),5)
  end if
])
fortext([Type], [DV MV], [
  forloop([Var],[1],[REP_NUMBER],[
    VERIFY_VARIABLE(Var,Type)
  ])
])

! Allocations and initializations.

call Initialize (dump_status)
call Initialize (consolidated_status)
```

```
! Open GMV file for writing.

unit = 19
if (this_is_IO_PE) then
  open (UNIT=unit, FILE=Filename, STATUS='new', IOSTAT=GMV_Status)
end if
call Broadcast (GMV_Status)
if (GMV_Status > 0) then
  if (this_is_IO_PE) then
    ifelse(COMPILER, Lahey, [
      call IOSTAT_MSG (GMV_Status, Error_Message)
      write (6,*) 'Dump_GMV_Multi_Mesh: IOSTAT message = ', &
                   TRIM(Error_Message)
    ])
    write (6,*) 'Dump_GMV_Multi_Mesh: GMV_Status = ', GMV_Status
    write (6,*) 'Dump_GMV_Multi_Mesh: File open error -- ', &
                'requested GMV file may already exist.'
  end if
  dump_status(1) = 'File Error'
end if

! Write GMV header.

if (this_is_IO_PE) write (unit,100) 'gmvinput ascii'

! Write GMV node coordinates after assembling on the IO PE.

call Initialize (Coordinates_Nodes_AV, Mesh%Node_Structure, &
                 2, 'Coordinates of Nodes', dump_status(2), &
                 Mesh%NDimensions)
call Initialize (Coordinates_Nodes_BNV, Mesh%NDimensions, &
                 Mesh%NNodes_total, dump_status(3))
Coordinates_Nodes_AV = Mesh%Coordinates_Nodes_DV
Coordinates_Nodes_BNV = Coordinates_Nodes_AV
if (this_is_IO_PE) then
  write (unit,100) 'nodev ', Mesh%NNodes_total
  select case (Mesh%NDimensions)
  case (1)
    do node = 1, Mesh%NNodes_total
      write (unit,101) Coordinates_Nodes_BNV(:,node), zero, zero
    end do
  case (2)
    do node = 1, Mesh%NNodes_total
      write (unit,101) Coordinates_Nodes_BNV(:,node), zero
    end do
  case (3)
    do node = 1, Mesh%NNodes_total
      write (unit,101) Coordinates_Nodes_BNV(:,node)
    end do
  end select
end if
call Finalize (Coordinates_Nodes_BNV, dump_status(4))
call Finalize (Coordinates_Nodes_AV, dump_status(5))
```

```
! Write GMV Cell Connectivity after assembling on the IO PE.

if (this_is_IO_PE) then
  Index_Size = Mesh%NCells_total
else
  Index_Size = 0
end if
call Initialize (Nodes_of_Cells_Index_Val_PE, Mesh%NCells_PE, &
                 Mesh%Nodes_per_Cell, dump_status(6))
call Initialize (Nodes_of_Cells_Index_Val_Total, Index_Size, &
                 Mesh%Nodes_per_Cell, dump_status(7))
Nodes_of_Cells_Index_Val_PE = Mesh%Nodes_of_Cells_Index
do node = 1, Mesh%Nodes_per_Cell
  call Assemble (Nodes_of_Cells_Index_Val_Total(:,node), &
                 Nodes_of_Cells_Index_Val_PE(:,node))
end do
if (this_is_IO_PE) then
  write (unit,100) 'cells ', Mesh%NCells_total
  do cell = 1, Mesh%NCells_total
    select case (Mesh%Shape)
    case ('Segmented')
      write (unit,100) 'line 2 ', Nodes_of_Cells_Index_Val_Total(cell,:)
    case ('Triangular')
      ! Numbering must be counter-clockwise for GMV.
      write (unit,100) 'tri 3 ', Nodes_of_Cells_Index_Val_Total(cell,:)
    case ('Quadrilateral')
      ! Numbering must be counter-clockwise for GMV.
      write (unit,100) 'quad 4 ', &
                       Nodes_of_Cells_Index_Val_Total(cell,1), &
                       Nodes_of_Cells_Index_Val_Total(cell,2), &
                       Nodes_of_Cells_Index_Val_Total(cell,4), &
                       Nodes_of_Cells_Index_Val_Total(cell,3)
    case ('Polygonal')
      VERIFY(.false.,0)   ! Not implemented yet.
    case ('Tetrahedral')
      ! An ordering is specified in the GMV docs. Not sure if
      ! to-be-implemented ordering in Caesar will comply.
      write (unit,100) 'tet 4 ', &
                       Nodes_of_Cells_Index_Val_Total(cell,:)
    case ('Hexahedral')
      ! GMV uses alternate node ordering.
      write (unit,100) 'hex 8 ', &
                       Nodes_of_Cells_Index_Val_Total(cell,1), &
                       Nodes_of_Cells_Index_Val_Total(cell,2), &
                       Nodes_of_Cells_Index_Val_Total(cell,4), &
                       Nodes_of_Cells_Index_Val_Total(cell,3), &
                       Nodes_of_Cells_Index_Val_Total(cell,5), &
                       Nodes_of_Cells_Index_Val_Total(cell,6), &
                       Nodes_of_Cells_Index_Val_Total(cell,8), &
                       Nodes_of_Cells_Index_Val_Total(cell,7)
    case ('Polyhedral')
      VERIFY(.false.,0)   ! Not implemented yet.
    end select
  end do
```

```
end if
call Finalize (Nodes_of_Cells_Index_Val_PE, dump_status(8))
call Finalize (Nodes_of_Cells_Index_Val_Total, dump_status(9))

! Write out variables.

if (this_is_IO_PE) write (unit,100) 'variable'

define([CALL_DUMP_GMV_VARIABLE],[
  pushdef([TYPE], [$2])
  ifelse(TYPE, [Mathematic], [
    pushdef([VARIABLE], [Variable$1_MV])
    pushdef([DUMPNUMBER], [10+$1])
  ],[
    pushdef([VARIABLE], [Variable$1_DV])
    pushdef([DUMPNUMBER], [10+REP_NUMBER+$1])
  ])
  pushdef([Dump_GMV_TYPE_Vector], expand(Dump_GMV_TYPE_Vector))
  if (PRESENT(VARIABLE)) then
    call Dump_GMV_TYPE_Vector (VARIABLE, Mesh, unit, &
                               dump_status(DUMPNUMBER))
  end if
  popdef([Dump_GMV_TYPE_Vector])
  popdef([DUMPNUMBER])
  popdef([VARIABLE])
  popdef([TYPE])
])
fortext([Type], [Mathematic Distributed], [
  forloop([Var],[1],[REP_NUMBER],[
    CALL_DUMP_GMV_VARIABLE(Var,Type)
  ])
])

if (this_is_IO_PE) write (unit,100) 'endvars'

! Write GMV closing statement and close GMV file.

if (this_is_IO_PE) then
  write (unit,100) 'endgmv'
  close (UNIT=unit, IOSTAT=GMV_Status)
end if
call Broadcast (GMV_Status)
if (GMV_Status > 0) dump_status(10) = 'File Error'

! Consolidate and handle status.

consolidated_status = dump_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
  VERIFY(Normal(consolidated_status), 5)
end if
call Finalize (consolidated_status)
```

```
    call Finalize (dump_status)

    ! Format statements.

100 format (a,5(:,i11))
101 format ((1pg15.8,4(:,1pg16.8)))

    ! Verify guarantees - none.

    return
  end subroutine Dump_GMV_Multi_Mesh
```

I.1.9 Dump_GMV DV and MV Vector Procedures

The main documentation of the Dump_GMV DV and MV Vector Procedures in § 14.1.9 on page 176 contains additional explanation of this code listing.

```
  define([DUMP_GMV_VARIABLE_ROUTINE],[
    subroutine Dump_GMV_$1_Vector (Variable, Mesh, unit, status)

      ! Input variables.

      type($1_Vector_type), intent(in) :: Variable    ! Variable to be output.
      type(Multi_Mesh_type), intent(in) :: Mesh       ! Mesh to be output.
      type(integer), intent(in) :: unit               ! GMV output unit.

      ! Output variable.

      type(Status_type), intent(out), optional :: status ! Exit status.

      ! Internal variables.

      type(integer) :: GMV_Locus_Number ! GMV Locus: 0-Cells, 1-Nodes, 2-Faces.
      type(integer) :: location         ! Location of the space character.
      type(character,name_length) :: Variable_Name  ! Name of the GMV variable.
      type(real,1) :: Variable_PE     ! BNV of the variable on each PE.
      type(real,1) :: Variable_Total  ! BNV of the total variable on the IO_PE.
      type(Status_type) :: consolidated_status       ! Consolidated Status.
      type(Status_type), dimension(4) :: dump_status  ! Status vector.

      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

      ! Verify requirements.

      VERIFY(Valid_State(Mesh),5)      ! Mesh is valid.
      VERIFY(Valid_State(Variable),5)  ! Variable is valid.

      ! Allocations and initializations.

      call Initialize (dump_status)
      call Initialize (consolidated_status)
```

```
! Get Variable Name and process to replace spaces with underscores.

Variable_Name = Name(Variable)
location = SCAN(TRIM(Variable_Name), " ")
do while (location /= 0)
  Variable_Name(location:location) = "_"
  location = SCAN(TRIM(Variable_Name), " ")
end do

! Toggle on Variable Locus to initialize temporaries.

select case (Locus(Variable))
case ("Cells")
  GMV_Locus_Number = 0
  call Initialize (Variable_PE, Mesh%NCells_PE, dump_status(1))
  call Initialize (Variable_Total, Mesh%NCells_Total, dump_status(2))
case ("Nodes")
  GMV_Locus_Number = 1
  call Initialize (Variable_PE, Mesh%NNodes_PE, dump_status(1))
  call Initialize (Variable_Total, Mesh%NNodes_Total, dump_status(2))
case ("Faces")
  GMV_Locus_Number = 2
  call Initialize (Variable_PE, Mesh%NFaces_PE, dump_status(1))
  call Initialize (Variable_Total, Mesh%NFaces_Total, dump_status(2))
  VERIFY(.false.,1) ! Face-based variables cannot be output to GMV until
                    ! the mesh is defined by faces instead of cells.
case default
  VERIFY(.false.,1) ! GMV variable output is only available for
                    ! mesh-based variables with a Locus of Cells,
                    ! Nodes or Faces.
end select

! Move data to the IO_PE and output.

Variable_PE = Variable
call Assemble (Variable_Total, Variable_PE)
if (this_is_IO_PE) then
  ! GMV limit is 32 characters.
  write (unit,100) TRIM(Variable_Name(:32)), GMV_Locus_Number
  write (unit,101) Variable_Total
end if

! Clean up temporary vectors.

call Finalize (Variable_PE, dump_status(3))
call Finalize (Variable_Total, dump_status(4))

! Consolidate and handle status.

consolidated_status = dump_status
if (PRESENT(status)) then
  WARN_IF(Error(consolidated_status), 5)
  status = consolidated_status
else
```

```
        VERIFY(Normal(consolidated_status), 5)
      end if
      call Finalize (consolidated_status)
      call Finalize (dump_status)

      ! Format statements.

100   format (a,5(:,i11))
101   format ((1pg15.8,4(:,1pg16.8)))

      ! Verify guarantees - none.

      return
    end subroutine Dump_GMV_$1_Vector
  ])
  fortext([Type], [Mathematic Distributed],[
    DUMP_GMV_VARIABLE_ROUTINE(Type)
  ])
```

I.1.10 Dump_XMGrace_Multi_Mesh Procedure

The main documentation of the Dump_XMGrace_Multi_Mesh Procedure in § 14.1.10 on page 177 contains additional explanation of this code listing.

```
  ! Set number of Mathematic Vectors and Distributed Vectors allowed
  ! in the call line. This can be changed here and it will change all
  ! necessary code to match. This makes use of the replicate m4 macros.
  ! Setting this to a number greater than 6 triggers errors on compilers
  ! that limit lines to 132 characters (like Absoft).

  define(REP_NUMBER, 6)

  subroutine Dump_XMGrace_Multi_Mesh (Filename, Mesh, Coordinate, &
    Xmin, Xmax, Ymin, Ymax, Zmin, Zmax &
    REP_ARGS([Variable[]i[]_MV]) &
    REP_ARGS([Variable[]i[]_DV]) &
    , status)

    ! Input variables.

    type(character,*), intent(in) :: Filename       ! Output filename.
    type(Multi_Mesh_type), intent(inout) :: Mesh    ! Mesh to be output.
    type(character,1), intent(in) :: Coordinate     ! X, Y or Z (for output).
    ! Output Mathematic_Vector variables.
    REP_DECLARE([type(Mathematic_Vector_type), optional], [Variable[]i[]_MV])
    ! Output Distributed_Vector variables.
    REP_DECLARE([type(Distributed_Vector_type), optional], [Variable[]i[]_DV])
    ! Limits on the points to be output.
    type(real), intent(in), optional :: Xmin, Xmax, Ymin, Ymax, Zmin, Zmax

    ! Output variable.
```

```
type(Status_type), optional :: status           ! Consolidated Status.

! Internal variables.

type(integer) :: XMGrace_Status                 ! XMGrace file open status.
type(integer) :: unit                           ! XMGrace output unit.
! Actual limits on the points to be output.
type(real) :: A_Xmin, A_Xmax, A_Ymin, A_Ymax, A_Zmin, A_Zmax
! Status vector.
type(Status_type), dimension(2+2*REP_NUMBER) :: dump_status
type(Status_type) :: consolidated_status           ! Consolidated Status.
ifelse(COMPILER, Lahey, [
  type(character,256) :: Error_Message ! String for compiler error message.
])

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Mesh),5)      ! Mesh is valid.
! Variable#_(DV|MV) is valid.
define([VERIFY_VARIABLE],[
  if (PRESENT(Variable$1_$2)) then
    VERIFY(Valid_State(Variable$1_$2),5)
  end if
])
fortext([Type], [DV MV], [
  forloop([Var],[1],[REP_NUMBER],[
    VERIFY_VARIABLE(Var,Type)
  ])
])

! Allocations and initializations.

call Initialize (dump_status)
call Initialize (consolidated_status)

! Set actual variables.

if (PRESENT(Xmin)) then
  A_Xmin = Xmin
else
  A_Xmin = -HUGE(one)
end if
if (PRESENT(Xmax)) then
  A_Xmax = Xmax
else
  A_Xmax = HUGE(one)
end if
if (PRESENT(Ymin)) then
  A_Ymin = Ymin
else
  A_Ymin = -HUGE(one)
end if
```

```
    if (PRESENT(Ymax)) then
      A_Ymax = Ymax
    else
      A_Ymax = HUGE(one)
    end if
    if (PRESENT(Zmin)) then
      A_Zmin = Zmin
    else
      A_Zmin = -HUGE(one)
    end if
    if (PRESENT(Zmax)) then
      A_Zmax = Zmax
    else
      A_Zmax = HUGE(one)
    end if

    ! Open XMGrace file for writing.

    unit = 19
    if (this_is_IO_PE) then
      open (UNIT=unit, FILE=Filename, STATUS='new', IOSTAT=XMGrace_Status)
    end if
    call Broadcast (XMGrace_Status)
    if (XMGrace_Status > 0) then
      if (this_is_IO_PE) then
        ifelse(COMPILER, Lahey, [
          call IOSTAT_MSG (XMGrace_Status, Error_Message)
          write (6,*) 'Dump_XMGrace_Multi_Mesh: IOSTAT message = ', &
                      TRIM(Error_Message)
        ])
        write (6,*) 'Dump_XMGrace_Multi_Mesh: XMGrace_Status = ', XMGrace_Status
        write (6,*) 'Dump_XMGrace_Multi_Mesh: File open error -- ', &
                    'requested XMGrace file may already exist.'
      end if
      dump_status(1) = 'File Error'
    end if

    ! Write out variables.

    define([CALL_DUMP_XMGrace_VARIABLE],[
      pushdef([TYPE], [$2])
      ifelse(TYPE, [Mathematic], [
        pushdef([VARIABLE], [Variable$1_MV])
        pushdef([DUMPNUMBER], [2+$1])
      ],[
        pushdef([VARIABLE], [Variable$1_DV])
        pushdef([DUMPNUMBER], [2+REP_NUMBER+$1])
      ])
      pushdef([Dump_XMGrace_TYPE_Vector], expand(Dump_XMGrace_TYPE_Vector))
      if (PRESENT(VARIABLE)) then
        call Dump_XMGrace_TYPE_Vector (VARIABLE, Mesh, Coordinate, unit, &
                                       A_Xmin, A_Xmax, &
                                       A_Ymin, A_Ymax, &
                                       A_Zmin, A_Zmax, &
```

```
                                       dump_status(DUMPNUMBER))
      end if
      popdef([Dump_XMGrace_TYPE_Vector])
      popdef([DUMPNUMBER])
      popdef([VARIABLE])
      popdef([TYPE])
    ])
    fortext([Type], [Mathematic Distributed], [
      forloop([Var],[1],[REP_NUMBER],[
        CALL_DUMP_XMGrace_VARIABLE(Var,Type)
      ])
    ])

    ! Close XMGrace file.

    if (this_is_IO_PE) then
      close (UNIT=unit, IOSTAT=XMGrace_Status)
    end if
    call Broadcast (XMGrace_Status)
    if (XMGrace_Status > 0) dump_status(2) = 'File Error'

    ! Consolidate and handle status.

    consolidated_status = dump_status
    if (PRESENT(status)) then
      WARN_IF(Error(consolidated_status), 5)
      status = consolidated_status
    else
      VERIFY(Normal(consolidated_status), 5)
    end if
    call Finalize (consolidated_status)
    call Finalize (dump_status)

    ! Format statements.

!100 format (a,5(:,i11))
!101 format ((1pg15.8,4(:,1pg16.8)))

    ! Verify guarantees - none.

    return
  end subroutine Dump_XMGrace_Multi_Mesh
```

I.1.11 Dump_XMGrace DV and MV Vector Procedures

The main documentation of the Dump_XMGrace DV and MV Vector Procedures in § 14.1.11 on page 178 contains additional explanation of this code listing.

```
  define([DUMP_XMGrace_VARIABLE_ROUTINE],[
    subroutine Dump_XMGrace_$1_Vector (Variable, Mesh, Coordinate, unit, &
                                       Xmin, Xmax, Ymin, Ymax, Zmin, Zmax, &
                                       status)
```

```
! Input variables.

type($1_Vector_type), intent(in) :: Variable    ! Variable to be output.
type(Multi_Mesh_type), intent(inout) :: Mesh    ! Mesh to be output.
type(character,1) :: Coordinate                 ! X, Y or Z (for output).
type(integer), intent(in) :: unit               ! XMGrace output unit.
! Limits on the points to be output.
type(real), intent(in) :: Xmin, Xmax, Ymin, Ymax, Zmin, Zmax

! Output variable.

type(Status_type), intent(out), optional :: status ! Exit status.

! Internal variables.

type(integer) :: XMGrace_Locus_Number ! XMGrace Locus: 0-Cells, 1-Nodes,
                                      ! 2-Faces.
type(real,2) :: Coordinates_Cells_PE  ! Coordinates of cells on the PE.
type(real,2) :: Coordinates_Cells_Total ! Coordinates of all cells.
type(integer) :: cell                 ! Cell number loop variable.
type(integer) :: dim                  ! Dimension loop variable.
type(real,1) :: Variable_PE     ! BNV of the variable on each PE.
type(real,1) :: Variable_Total  ! BNV of the total variable on the IO_PE.
type(Status_type) :: consolidated_status       ! Consolidated Status.
type(Status_type), dimension(8) :: dump_status  ! Status vector.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Mesh),5)      ! Mesh is valid.
VERIFY(Valid_State(Variable),5)  ! Variable is valid.

! Allocations and initializations.

call Initialize (dump_status)
call Initialize (consolidated_status)

! Toggle on Variable Locus to initialize temporaries.

select case (Locus(Variable))
case ("Cells")
  XMGrace_Locus_Number = 0
  call Initialize (Variable_PE, Mesh%NCells_PE, dump_status(1))
  call Initialize (Variable_Total, Mesh%NCells_Total, dump_status(2))
  call Initialize (Coordinates_Cells_PE, Mesh%NDimensions, &
                   Mesh%NCells_PE, dump_status(3))
  call Initialize (Coordinates_Cells_Total, Mesh%NDimensions, &
                   Mesh%NCells_Total, dump_status(4))
  call Get_Coordinates_Cells (Coordinates_Cells_PE, Mesh)
  do dim = 1, Mesh%NDimensions
    call Assemble (Coordinates_Cells_Total(dim,:), &
                   Coordinates_Cells_PE(dim,:))
```

```
  end do
case ("Nodes")
  XMGrace_Locus_Number = 1
  call Initialize (Variable_PE, Mesh%NNodes_PE, dump_status(1))
  call Initialize (Variable_Total, Mesh%NNodes_Total, dump_status(2))
  VERIFY(.false.,1) ! Node-based variables cannot be output to XMGrace
                    ! until a Get_Coordinates_Nodes procedure is written.
case ("Faces")
  XMGrace_Locus_Number = 2
  call Initialize (Variable_PE, Mesh%NFaces_PE, dump_status(1))
  call Initialize (Variable_Total, Mesh%NFaces_Total, dump_status(2))
  VERIFY(.false.,1) ! Face-based variables cannot be output to XMGrace
                    ! until the mesh is defined by faces instead of cells.
case default
  VERIFY(.false.,1) ! XMGrace variable output is only available for
                    ! mesh-based variables with a Locus of Cells,
                    ! Nodes or Faces.
end select

! Move data to the IO_PE and output.

Variable_PE = Variable
call Assemble (Variable_Total, Variable_PE)
if (this_is_IO_PE .AND. XMGrace_Locus_Number==0) then
  write (unit,*)
  write (unit,*) '#     ', Coordinate, '          ', TRIM(Name(Variable))
  do cell = 1, Mesh%NCells_Total
    ! InInterval wouldn't work here. Not sure why, maybe explore later.
    if (Coordinates_Cells_Total(1,cell) >= Xmin .AND. &
        Coordinates_Cells_Total(1,cell) <= Xmax .AND. &
        Coordinates_Cells_Total(2,cell) >= Ymin .AND. &
        Coordinates_Cells_Total(2,cell) <= Ymax .AND. &
        Coordinates_Cells_Total(3,cell) >= Zmin .AND. &
        Coordinates_Cells_Total(3,cell) <= Zmax) then
      select case (Coordinate)
      case ("X")
        write (unit,*) Coordinates_Cells_Total(1,cell), &
                       Variable_Total(cell)
      case ("Y")
        write (unit,*) Coordinates_Cells_Total(2,cell), &
                       Variable_Total(cell)
      case ("Z")
        write (unit,*) Coordinates_Cells_Total(3,cell), &
                       Variable_Total(cell)
      end select
    end if
  end do
end if

! Clean up temporary vectors.

call Finalize (Variable_PE, dump_status(5))
call Finalize (Variable_Total, dump_status(6))
if (XMGrace_Locus_Number==0) then
```

```
        call Finalize (Coordinates_Cells_PE, dump_status(7))
        call Finalize (Coordinates_Cells_Total, dump_status(8))
      end if

      ! Consolidate and handle status.

      consolidated_status = dump_status
      if (PRESENT(status)) then
        WARN_IF(Error(consolidated_status), 5)
        status = consolidated_status
      else
        VERIFY(Normal(consolidated_status), 5)
      end if
      call Finalize (consolidated_status)
      call Finalize (dump_status)

      ! Verify guarantees - none.

      return
    end subroutine Dump_XMGrace_$1_Vector
  ])
  fortext([Type], [Mathematic Distributed],[
    DUMP_XMGrace_VARIABLE_ROUTINE(Type)
  ])
```

I.1.12 Get_Area_Faces_of_Cells_Multi_Mesh Procedure

The main documentation of the Get_Area_Faces_of_Cells_Multi_Mesh Procedure in § 14.1.12 on page 178 contains additional explanation of this code listing.

```
  subroutine Get_Area_Faces_of_Cells_MMesh (Area_Faces_of_Cells, Mesh)

    ! Input variable.

    type(Multi_Mesh_type), intent(inout) :: Mesh   ! Mesh object.

    ! Input/Output variable.

    type(real,3) :: Area_Faces_of_Cells ! Area_Faces_of_Cells BNV.

    ! Internal variables.

    type(integer) :: d      ! Dimension loop parameter.
    ! BNV of coordinates of nodes of cells.
    type(real,3) :: Coordinates_Nodes_of_Cells

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)                       ! Mesh is valid.
    VERIFY(Valid_State(Area_Faces_of_Cells),5)  ! Area_Faces_of_Cells is valid.
```

```
! Area_Faces_of_Cells has correct dimensions.
VERIFY(SIZE(Area_Faces_of_Cells,1) == Mesh%NDimensions,5)
VERIFY(SIZE(Area_Faces_of_Cells,2) == Mesh%NCells_PE,5)
VERIFY(SIZE(Area_Faces_of_Cells,3) == Mesh%Faces_per_Cell,5)!Not for polys.

! Get Coordinates of the Nodes if needed.

if (Mesh%Uniformity == "Nonuniform") then

  call Initialize (Coordinates_Nodes_of_Cells, Mesh%NDimensions, &
                   Mesh%NCells_PE, Mesh%Nodes_per_Cell)
  call Get_Coordinates_Nodes_of_Cells (Coordinates_Nodes_of_Cells, Mesh)

end if

! Set the vector areas for each face.

Area_Faces_of_Cells = zero
if (Mesh%Uniformity == "Uniform") then

  ! For a uniform mesh, all cells have the same
  ! three areas, plus or minus.
  do d = 1, Mesh%NDimensions
    Area_Faces_of_Cells(d,:,2*d-1) = -Mesh%Area_All_Faces(d)
    Area_Faces_of_Cells(d,:,2*d  ) =  Mesh%Area_All_Faces(d)
  end do

else if (Mesh%Orthogonality == "Orthogonal") then

  ! For an orthogonal mesh, the face areas may be calculated from the
  ! product of 0, 1 or 2 orthogonal edge lengths. First, set positive
  ! (+X,+Y,+Z) face areas.
  select case (Mesh%NDimensions)
  case (1)
    ! Suppressed Y, Z.
    Area_Faces_of_Cells(:,:,2) =  one     ! Right (+X) Face.
  case (2)
    ! Suppressed Z.
    Area_Faces_of_Cells(1,:,2) = &        ! Right (+X) Face.
      ( Coordinates_Nodes_of_Cells(2,:,3) &     ! Delta Y -
      - Coordinates_Nodes_of_Cells(2,:,1) )     ! Nodes 1 & 3
    Area_Faces_of_Cells(2,:,4) = &        ! Back (+Y) Face.
      ( Coordinates_Nodes_of_Cells(1,:,2) &     ! Delta X -
      - Coordinates_Nodes_of_Cells(1,:,1) )     ! Nodes 1 & 2
  case (3)
    Area_Faces_of_Cells(1,:,2) = &        ! Right (+X) Face.
      ( Coordinates_Nodes_of_Cells(2,:,3) &     ! Delta Y -
      - Coordinates_Nodes_of_Cells(2,:,1) ) * & ! Nodes 1 & 3
      ( Coordinates_Nodes_of_Cells(3,:,5) &     ! Delta Z -
      - Coordinates_Nodes_of_Cells(3,:,1) )     ! Nodes 1 & 5
    Area_Faces_of_Cells(2,:,4) = &        ! Back (+Y) Face.
      ( Coordinates_Nodes_of_Cells(1,:,2) &     ! Delta X -
      - Coordinates_Nodes_of_Cells(1,:,1) ) * & ! Nodes 1 & 2
      ( Coordinates_Nodes_of_Cells(3,:,5) &     ! Delta Z -
```

```
        - Coordinates_Nodes_of_Cells(3,:,1) )     ! Nodes 1 & 5
      Area_Faces_of_Cells(3,:,6) = &        ! Top (+Z) Face.
        ( Coordinates_Nodes_of_Cells(1,:,2) &     ! Delta X -
        - Coordinates_Nodes_of_Cells(1,:,1) ) * & ! Nodes 1 & 2
        ( Coordinates_Nodes_of_Cells(2,:,3) &     ! Delta Y -
        - Coordinates_Nodes_of_Cells(2,:,1) )     ! Nodes 1 & 3
    end select
    ! Set opposing (-X,-Y,-Z) faces to negative of the positive faces.
    do d = 1, Mesh%NDimensions
      Area_Faces_of_Cells(d,:,2*d-1) = -Area_Faces_of_Cells(d,:,2*d)
    end do

  else
    ! Coding for other mesh types not implemented yet.
    VERIFY(.false.,1)
  end if

  ! Finalize temporaries.

  if (Mesh%Uniformity == "Nonuniform") then
    call Finalize (Coordinates_Nodes_of_Cells)
  end if

  ! Verify guarantees.

  VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
  VERIFY(Valid_State(Area_Faces_of_Cells),5) ! Area_Faces_of_Cells is valid.

  return
end subroutine Get_Area_Faces_of_Cells_MMesh
```

I.1.13 Get_Coordinates_Cells_Multi_Mesh Procedure

The main documentation of the Get_Coordinates_Cells_Multi_Mesh Procedure in § 14.1.13 on page 179 contains additional explanation of this code listing.

```
subroutine Get_Coordinates_Cells_MMesh (Coordinates_Cells, Mesh)

  ! Input variable.

  type(Multi_Mesh_type), intent(inout) :: Mesh   ! Mesh object.

  ! Input/Output variable.

  type(real,2) :: Coordinates_Cells ! Coordinates_Cells BNV.

  !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

  ! Verify requirements.

  VERIFY(Valid_State(Mesh),5)                       ! Mesh is valid.
  ! Coordinates_Cells is valid.
```

```
    VERIFY(Valid_State(Coordinates_Cells),5)
    ! Coordinates_Cells has correct dimensions.
    VERIFY(SIZE(Coordinates_Cells,1) == Mesh%NDimensions,5)
    VERIFY(SIZE(Coordinates_Cells,2) == Mesh%NCells_PE,5)

    ! Initializations.

    if (.not.Initialized(Mesh%Coordinates_Cells_DV)) then
      call Initialize (Mesh%Coordinates_Cells_DV, Mesh%Cell_Structure, 2, &
                       dim1=Mesh%NDimensions)
    end if
    if (.not.Initialized(Mesh%Coordinates_Nodes_of_Cells_CA)) then
      call Initialize (Mesh%Coordinates_Nodes_of_Cells_CA, &
                       Mesh%Nodes_of_Cells_Index, 2, &
                       dim1=Mesh%NDimensions)
    end if

    ! Gather node coordinates to cells.

    Mesh%Coordinates_Nodes_of_Cells_CA = Mesh%Coordinates_Nodes_DV

    ! Calculate cell center coordinates (= average of the node coordinates).

    call Combine_with_Average (Mesh%Coordinates_Cells_DV, &
                               Mesh%Coordinates_Nodes_of_Cells_CA)
    Coordinates_Cells = Mesh%Coordinates_Cells_DV

    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
    ! Coordinates_Cells is valid.
    VERIFY(Valid_State(Coordinates_Cells),5)

    return
  end subroutine Get_Coordinates_Cells_MMesh
```

I.1.14 Get_Coordinates_Cells_of_Cells_Multi_Mesh Procedure

The main documentation of the Get_Coordinates_Cells_of_Cells_Multi_Mesh Procedure in § 14.1.14 on page 179 contains additional explanation of this code listing.

```
  subroutine Get_Coordinates_CoC_MMesh (Coordinates_Cells_of_Cells, Mesh)

    ! Input variable.

    type(Multi_Mesh_type), intent(inout) :: Mesh   ! Mesh object.

    ! Input/Output variable.

    ! Coordinates_Cells_of_Cells BNV.
    type(real,3) :: Coordinates_Cells_of_Cells
```

```
! Internal variables.

type(integer,2) :: Flag_Faces_of_Cells ! Mesh boundary flags.

!~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

! Verify requirements.

VERIFY(Valid_State(Mesh),5)                       ! Mesh is valid.
! Coordinates_Cells_of_Cells is valid.
VERIFY(Valid_State(Coordinates_Cells_of_Cells),5)
! Coordinates_Cells_of_Cells has correct dimensions.
VERIFY(SIZE(Coordinates_Cells_of_Cells,1) == Mesh%NDimensions,5)
VERIFY(SIZE(Coordinates_Cells_of_Cells,2) == Mesh%NCells_PE,5)
! Following is not true for polys.
VERIFY(SIZE(Coordinates_Cells_of_Cells,3) == Mesh%Faces_per_Cell,5)

! Initializations.

if (.not.Initialized(Mesh%Coordinates_Cells_DV)) then
  call Initialize (Mesh%Coordinates_Cells_DV, Mesh%Cell_Structure, 2, &
                   dim1=Mesh%NDimensions)
end if
if (.not.Initialized(Mesh%Coordinates_Nodes_of_Cells_CA)) then
  call Initialize (Mesh%Coordinates_Nodes_of_Cells_CA, &
                   Mesh%Nodes_of_Cells_Index, 2, &
                   dim1=Mesh%NDimensions)
end if
if (.not.Initialized(Mesh%Coordinates_Cells_of_Cells_CA)) then
  call Initialize (Mesh%Coordinates_Cells_of_Cells_CA, &
                   Mesh%Cells_of_Cells_Index, 2, &
                   dim1=Mesh%NDimensions)
end if

! Gather node coordinates to cells.

Mesh%Coordinates_Nodes_of_Cells_CA = Mesh%Coordinates_Nodes_DV

! Calculate cell center coordinates (= average of the node coordinates).

call Combine_with_Average (Mesh%Coordinates_Cells_DV, &
                           Mesh%Coordinates_Nodes_of_Cells_CA)

! Collect coordinates from "other" cells to cells.

Mesh%Coordinates_Cells_of_Cells_CA = Mesh%Coordinates_Cells_DV
Coordinates_Cells_of_Cells = Mesh%Coordinates_Cells_of_Cells_CA

! Note that the "other" cell coordinates are now correct throughout
! the center of the mesh, but incorrect across a boundary face.
!
! For nonorthogonal meshes, the "other" cell coordinates across a
! boundary face should never be used.
!
```

```
    ! For orthogonal meshes, the "other" cell coordinates across a boundary
    ! face are set to a value which will allow correct modeling of periodic
    ! boundary conditions. The "other" cell index was already set to the
    ! periodic next cell, so the current coordinate values are for cells on
    ! the far side of the problem. This is fixed by adding/subtracting the
    ! problem lengths.

    if (Mesh%Orthogonality == "Orthogonal") then
      call Initialize (Flag_Faces_of_Cells, NCells_PE(Mesh), &
                       Mesh%Faces_per_Cell)
      call Get_Flag_Faces_of_Cells (Flag_Faces_of_Cells, Mesh)

      ! Fix the edges, since they wrapped around incorrectly.
      ! This is an assumption that is only valid for ortho cells.

      where (Flag_Faces_of_Cells == 1)      ! Left (-x) Face.
        Coordinates_Cells_of_Cells(1,:,:) = &
          Coordinates_Cells_of_Cells(1,:,:) - &
          Mesh%Lengths(1)
      elsewhere (Flag_Faces_of_Cells == 2) ! Right (+x) Face.
        Coordinates_Cells_of_Cells(1,:,:) = &
          Coordinates_Cells_of_Cells(1,:,:) + &
          Mesh%Lengths(1)
      elsewhere (Flag_Faces_of_Cells == 3) ! Front (-y) Face.
        Coordinates_Cells_of_Cells(2,:,:) = &
          Coordinates_Cells_of_Cells(2,:,:) - &
          Mesh%Lengths(2)
      elsewhere (Flag_Faces_of_Cells == 4) ! Back (+y) Face.
        Coordinates_Cells_of_Cells(2,:,:) = &
          Coordinates_Cells_of_Cells(2,:,:) + &
          Mesh%Lengths(2)
      elsewhere (Flag_Faces_of_Cells == 5) ! Bottom (-z) Face.
        Coordinates_Cells_of_Cells(3,:,:) = &
          Coordinates_Cells_of_Cells(3,:,:) - &
          Mesh%Lengths(3)
      elsewhere (Flag_Faces_of_Cells == 6) ! Top (+z) Face.
        Coordinates_Cells_of_Cells(3,:,:) = &
          Coordinates_Cells_of_Cells(3,:,:) + &
          Mesh%Lengths(3)
      end where
      call Finalize (Flag_Faces_of_Cells)
    end if

    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
    ! Coordinates_Cells_of_Cells is valid.
    VERIFY(Valid_State(Coordinates_Cells_of_Cells),5)

    return
  end subroutine Get_Coordinates_CoC_MMesh
```

I.1.15 Get_Coordinates_Faces_of_Cells_Multi_Mesh Procedure

The main documentation of the Get_Coordinates_Faces_of_Cells_Multi_Mesh Procedure in § 14.1.15 on page 179 contains additional explanation of this code listing.

```
  subroutine Get_Coordinates_FoC_MMesh (Coordinates_Faces_of_Cells, Mesh)

    ! Input variable.

    type(Multi_Mesh_type), intent(inout) :: Mesh   ! Mesh object.

    ! Input/Output variable.

    ! Coordinates_Faces_of_Cells BNV.
    type(real,3) :: Coordinates_Faces_of_Cells

    ! Internal variables.

    ! BNV of coordinates of nodes of cells.
    type(real,3) :: Coordinates_Nodes_of_Cells

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)                        ! Mesh is valid.
    ! Coordinates_Faces_of_Cells is valid.
    VERIFY(Valid_State(Coordinates_Faces_of_Cells),5)
    ! Coordinates_Faces_of_Cells has correct dimensions.
    VERIFY(SIZE(Coordinates_Faces_of_Cells,1) == Mesh%NDimensions,5)
    VERIFY(SIZE(Coordinates_Faces_of_Cells,2) == Mesh%NCells_PE,5)
    ! Following is not true for polys.
    VERIFY(SIZE(Coordinates_Faces_of_Cells,3) == Mesh%Faces_per_Cell,5)

    ! Get Coordinates of the Nodes.

    call Initialize (Coordinates_Nodes_of_Cells, Mesh%NDimensions, &
                     Mesh%NCells_PE, Mesh%Nodes_per_Cell)
    call Get_Coordinates_Nodes_of_Cells (Coordinates_Nodes_of_Cells, Mesh)

    ! Set the coordinates for each face.

    ! Note: probably should generalize this to an
    !       Average_Nodes_to_Faces procedure later.

    Coordinates_Faces_of_Cells = zero
    if (Mesh%Shape == "Segmented") then

      Coordinates_Faces_of_Cells(:,:,:) = Coordinates_Nodes_of_Cells(:,:,:)

    else if (Mesh%Shape == "Quadrilateral") then

      Coordinates_Faces_of_Cells(:,:,1) = &
```

```
    half*(Coordinates_Nodes_of_Cells(:,:,1) + &
          Coordinates_Nodes_of_Cells(:,:,3))
  Coordinates_Faces_of_Cells(:,:,2) = &
    half*(Coordinates_Nodes_of_Cells(:,:,2) + &
          Coordinates_Nodes_of_Cells(:,:,4))
  Coordinates_Faces_of_Cells(:,:,3) = &
    half*(Coordinates_Nodes_of_Cells(:,:,1) + &
          Coordinates_Nodes_of_Cells(:,:,2))
  Coordinates_Faces_of_Cells(:,:,4) = &
    half*(Coordinates_Nodes_of_Cells(:,:,3) + &
          Coordinates_Nodes_of_Cells(:,:,4))

else if (Mesh%Shape == "Hexahedral") then

  Coordinates_Faces_of_Cells(:,:,1) = &
    fourth*(Coordinates_Nodes_of_Cells(:,:,1) + &
            Coordinates_Nodes_of_Cells(:,:,3) + &
            Coordinates_Nodes_of_Cells(:,:,5) + &
            Coordinates_Nodes_of_Cells(:,:,7))
  Coordinates_Faces_of_Cells(:,:,2) = &
    fourth*(Coordinates_Nodes_of_Cells(:,:,2) + &
            Coordinates_Nodes_of_Cells(:,:,4) + &
            Coordinates_Nodes_of_Cells(:,:,6) + &
            Coordinates_Nodes_of_Cells(:,:,8))
  Coordinates_Faces_of_Cells(:,:,3) = &
    fourth*(Coordinates_Nodes_of_Cells(:,:,1) + &
            Coordinates_Nodes_of_Cells(:,:,2) + &
            Coordinates_Nodes_of_Cells(:,:,5) + &
            Coordinates_Nodes_of_Cells(:,:,6))
  Coordinates_Faces_of_Cells(:,:,4) = &
    fourth*(Coordinates_Nodes_of_Cells(:,:,3) + &
            Coordinates_Nodes_of_Cells(:,:,4) + &
            Coordinates_Nodes_of_Cells(:,:,7) + &
            Coordinates_Nodes_of_Cells(:,:,8))
  Coordinates_Faces_of_Cells(:,:,5) = &
    fourth*(Coordinates_Nodes_of_Cells(:,:,1) + &
            Coordinates_Nodes_of_Cells(:,:,2) + &
            Coordinates_Nodes_of_Cells(:,:,3) + &
            Coordinates_Nodes_of_Cells(:,:,4))
  Coordinates_Faces_of_Cells(:,:,6) = &
    fourth*(Coordinates_Nodes_of_Cells(:,:,5) + &
            Coordinates_Nodes_of_Cells(:,:,6) + &
            Coordinates_Nodes_of_Cells(:,:,7) + &
            Coordinates_Nodes_of_Cells(:,:,8))

else
  ! Coding for other mesh types not implemented yet.
  VERIFY(.false.,1)
end if

! Finalize temporaries.

call Finalize (Coordinates_Nodes_of_Cells)
```

```
    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
    ! Coordinates_Faces_of_Cells is valid.
    VERIFY(Valid_State(Coordinates_Faces_of_Cells),5)

    return
  end subroutine Get_Coordinates_FoC_MMesh
```

I.1.16 Get_Coordinates_Nodes_of_Cells_Multi_Mesh Procedure

The main documentation of the Get_Coordinates_Nodes_of_Cells_Multi_Mesh Procedure in § 14.1.16 on page 180 contains additional explanation of this code listing.

```
  subroutine Get_Coordinates_NoC_MMesh (Coordinates_Nodes_of_Cells, Mesh)

    ! Input variable.

    type(Multi_Mesh_type), intent(inout) :: Mesh   ! Mesh object.

    ! Input/Output variable.

    ! Coordinates_Nodes_of_Cells BNV.
    type(real,3) :: Coordinates_Nodes_of_Cells

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)                       ! Mesh is valid.
    ! Coordinates_Cells is valid.
    VERIFY(Valid_State(Coordinates_Nodes_of_Cells),5)
    ! Coordinates_Cells has correct dimensions.
    VERIFY(SIZE(Coordinates_Nodes_of_Cells,1) == Mesh%NDimensions,5)
    VERIFY(SIZE(Coordinates_Nodes_of_Cells,2) == Mesh%NCells_PE,5)
    VERIFY(SIZE(Coordinates_Nodes_of_Cells,3) == Mesh%Nodes_per_Cell,5)

    ! Initializations.

    if (.not.Initialized(Mesh%Coordinates_Nodes_of_Cells_CA)) then
      call Initialize (Mesh%Coordinates_Nodes_of_Cells_CA, &
                       Mesh%Nodes_of_Cells_Index, 2, &
                       dim1=Mesh%NDimensions)
    end if

    ! Gather node coordinates to cells.

    Mesh%Coordinates_Nodes_of_Cells_CA = Mesh%Coordinates_Nodes_DV
    Coordinates_Nodes_of_Cells = Mesh%Coordinates_Nodes_of_Cells_CA

    ! Verify guarantees.
```

```
    VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
    ! Coordinates_Cells is valid.
    VERIFY(Valid_State(Coordinates_Nodes_of_Cells),5)

    return
  end subroutine Get_Coordinates_NoC_MMesh
```

I.1.17 Get_DeltaR21_Cells_of_Cells_Multi_Mesh Procedure

The main documentation of the Get_DeltaR21_Cells_of_Cells_Multi_Mesh Procedure in § 14.1.17 on page 180 contains additional explanation of this code listing.

```
  subroutine Get_DeltaR21_C_of_C_MMesh (DeltaR21_Cells_of_Cells, Mesh)

    ! Input variable.

    type(Multi_Mesh_type), intent(inout) :: Mesh   ! Mesh object.

    ! Input/Output variable.

    type(real,2) :: DeltaR21_Cells_of_Cells ! DeltaR21_Cells_of_Cells BNV.

    ! Internal variables.

    type(integer) :: cell, other_cell        ! Loop variables.
    type(real,2) :: Coordinates_Cells        ! Coordinates of the cell centers.
    type(real,3) :: Coordinates_Cells_of_Cells ! Coordinates of other cells.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)                       ! Mesh is valid.
    ! DeltaR21_Cells_of_Cells is valid.
    VERIFY(Valid_State(DeltaR21_Cells_of_Cells),5)
    ! DeltaR21_Cells_of_Cells has correct dimensions.
    VERIFY(SIZE(DeltaR21_Cells_of_Cells,1) == Mesh%NCells_PE,5)
    VERIFY(SIZE(DeltaR21_Cells_of_Cells,2) == Mesh%Faces_per_Cell,5)

    ! Get cell and face coordinates.

    call Initialize (Coordinates_Cells, Mesh%NDimensions, Mesh%NCells_PE)
    call Get_Coordinates_Cells (Coordinates_Cells, Mesh)

    call Initialize (Coordinates_Cells_of_Cells, Mesh%NDimensions, &
                     Mesh%NCells_PE, Mesh%Faces_per_Cell)
    call Get_Coordinates_Cells_of_Cells (Coordinates_Cells_of_Cells, Mesh)

    ! Calculate absolute distance from cell center to other cells (across
    ! the faces),
    !
    !   DeltaR21 = | R_1 - R_2 |.
```

```
    do cell = 1, Mesh%NCells_PE
      do other_cell = 1, Mesh%Faces_per_Cell
        DeltaR21_Cells_of_Cells(cell,other_cell) = &
          SQRT(DOT_PRODUCT( &
            Coordinates_Cells(:,cell) - &
            Coordinates_Cells_of_Cells(:,cell,other_cell), &
            Coordinates_Cells(:,cell) - &
            Coordinates_Cells_of_Cells(:,cell,other_cell)))
      end do
    end do

    ! Finalizations.

    call Finalize (Coordinates_Cells)
    call Finalize (Coordinates_Cells_of_Cells)

    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
    ! DeltaR21_Cells_of_Cells is valid.
    VERIFY(Valid_State(DeltaR21_Cells_of_Cells),5)

    return
  end subroutine Get_DeltaR21_C_of_C_MMesh
```

I.1.18 Get_DeltaR1f_Cells_of_Cells_Multi_Mesh Procedure

The main documentation of the Get_DeltaR1f_Cells_of_Cells_Multi_Mesh Procedure in § 14.1.18 on page 180 contains additional explanation of this code listing.

```
  subroutine Get_DeltaR1f_C_of_C_MMesh (DeltaR1f_Cells_of_Cells, Mesh)

    ! Input variable.

    type(Multi_Mesh_type), intent(inout) :: Mesh   ! Mesh object.

    ! Input/Output variable.

    type(real,2) :: DeltaR1f_Cells_of_Cells ! DeltaR1f_Cells_of_Cells BNV.

    ! Internal variables.

    type(integer) :: cell, other_cell        ! Loop variables.
    type(real,2) :: Coordinates_Cells        ! Coordinates of the cell centers.
    type(real,3) :: Coordinates_Faces_of_Cells ! Coordinates of face centers.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)                       ! Mesh is valid.
```

```
    ! DeltaR1f_Cells_of_Cells is valid.
    VERIFY(Valid_State(DeltaR1f_Cells_of_Cells),5)
    ! DeltaR1f_Cells_of_Cells has correct dimensions.
    VERIFY(SIZE(DeltaR1f_Cells_of_Cells,1) == Mesh%NCells_PE,5)
    VERIFY(SIZE(DeltaR1f_Cells_of_Cells,2) == Mesh%Faces_per_Cell,5)

    ! Get cell and face coordinates.

    call Initialize (Coordinates_Cells, Mesh%NDimensions, Mesh%NCells_PE)
    call Get_Coordinates_Cells (Coordinates_Cells, Mesh)

    call Initialize (Coordinates_Faces_of_Cells, Mesh%NDimensions, &
                     Mesh%NCells_PE, Mesh%Faces_per_Cell)
    call Get_Coordinates_Faces_of_Cells (Coordinates_Faces_of_Cells, Mesh)

    ! Calculate absolute distance from cell center to all faces,
    !
    !   DeltaR1f = | R_f - R_1 |.

    do cell = 1, Mesh%NCells_PE
      do other_cell = 1, Mesh%Faces_per_Cell
        DeltaR1f_Cells_of_Cells(cell,other_cell) = &
          SQRT(DOT_PRODUCT( &
            Coordinates_Faces_of_Cells(:,cell,other_cell) - &
            Coordinates_Cells(:,cell), &
            Coordinates_Faces_of_Cells(:,cell,other_cell) - &
            Coordinates_Cells(:,cell)))
      end do
    end do

    ! Finalizations.

    call Finalize (Coordinates_Cells)
    call Finalize (Coordinates_Faces_of_Cells)

    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
    ! DeltaR1f_Cells_of_Cells is valid.
    VERIFY(Valid_State(DeltaR1f_Cells_of_Cells),5)

    return
  end subroutine Get_DeltaR1f_C_of_C_MMesh
```

I.1.19 Get_DeltaR2f_Cells_of_Cells_Multi_Mesh Procedure

The main documentation of the Get_DeltaR2f_Cells_of_Cells_Multi_Mesh Procedure in § 14.1.19 on page 181 contains additional explanation of this code listing.

```
  subroutine Get_DeltaR2f_C_of_C_MMesh (DeltaR2f_Cells_of_Cells, Mesh)

    ! Input variable.
```

```
    type(Multi_Mesh_type), intent(inout) :: Mesh   ! Mesh object.

    ! Input/Output variable.

    type(real,2) :: DeltaR2f_Cells_of_Cells ! DeltaR2f_Cells_of_Cells BNV.

    ! Internal variables.

    type(integer) :: cell, other_cell          ! Loop variables.
    type(real,3) :: Coordinates_Cells_of_Cells ! Coordinates of other cells.
    type(real,3) :: Coordinates_Faces_of_Cells ! Coordinates of face centers.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)                        ! Mesh is valid.
    ! DeltaR2f_Cells_of_Cells is valid.
    VERIFY(Valid_State(DeltaR2f_Cells_of_Cells),5)
    ! DeltaR2f_Cells_of_Cells has correct dimensions.
    VERIFY(SIZE(DeltaR2f_Cells_of_Cells,1) == Mesh%NCells_PE,5)
    VERIFY(SIZE(DeltaR2f_Cells_of_Cells,2) == Mesh%Faces_per_Cell,5)

    ! Get other_cell and face coordinates.

    call Initialize (Coordinates_Cells_of_Cells, Mesh%NDimensions, &
                     Mesh%NCells_PE, Mesh%Faces_per_Cell)
    call Get_Coordinates_Cells_of_Cells (Coordinates_Cells_of_Cells, Mesh)

    call Initialize (Coordinates_Faces_of_Cells, Mesh%NDimensions, &
                     Mesh%NCells_PE, Mesh%Faces_per_Cell)
    call Get_Coordinates_Faces_of_Cells (Coordinates_Faces_of_Cells, Mesh)

    ! Calculate absolute distance from cell center across the face (the
    ! other_cell) to the face,
    !
    !   DeltaR2f = | R_f - R_2 |.

    do cell = 1, Mesh%NCells_PE
      do other_cell = 1, Mesh%Faces_per_Cell
        DeltaR2f_Cells_of_Cells(cell,other_cell) = &
          SQRT(DOT_PRODUCT( &
            Coordinates_Faces_of_Cells(:,cell,other_cell) - &
            Coordinates_Cells_of_Cells(:,cell,other_cell), &
            Coordinates_Faces_of_Cells(:,cell,other_cell) - &
            Coordinates_Cells_of_Cells(:,cell,other_cell)))
      end do
    end do

    ! Finalizations.

    call Finalize (Coordinates_Cells_of_Cells)
    call Finalize (Coordinates_Faces_of_Cells)
```

```
    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
    ! DeltaR2f_Cells_of_Cells is valid.
    VERIFY(Valid_State(DeltaR2f_Cells_of_Cells),5)

    return
  end subroutine Get_DeltaR2f_C_of_C_MMesh
```

I.1.20 Get_Flag_Faces_of_Cells_Multi_Mesh Procedure

The main documentation of the Get_Flag_Faces_of_Cells_Multi_Mesh Procedure in § 14.1.20 on page 181 contains additional explanation of this code listing.

```
  subroutine Get_Flag_Faces_of_Cells_MMesh (Flag_Faces_of_Cells, Mesh)

    ! Input variable.

    type(Multi_Mesh_type), intent(in) :: Mesh   ! Mesh object.

    ! Input/Output variable.

    type(integer,2) :: Flag_Faces_of_Cells ! Flag_Faces_of_Cells BNV.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)                       ! Mesh is valid.
    ! Flag_Faces_of_Cells is valid.
    VERIFY(Valid_State(Flag_Faces_of_Cells),5)
    ! Flag_Faces_of_Cells has correct dimensions.
    VERIFY(SIZE(Flag_Faces_of_Cells,1) == Mesh%NCells_PE,5)
    VERIFY(SIZE(Flag_Faces_of_Cells,2) == Mesh%Faces_per_Cell,5) ! !=polys.

    ! Set the Flag for each face.

    if (Mesh%Orthogonality == "Orthogonal") then
      Flag_Faces_of_Cells = Mesh%Flag_Faces_of_Cells
    else
      ! Coding for other mesh types not implemented yet.
      VERIFY(.false.,1)
    end if

    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)                ! Mesh is valid.
    ! Flag_Faces_of_Cells is valid.
    VERIFY(Valid_State(Flag_Faces_of_Cells),5)

    return
```

```
end subroutine Get_Flag_Faces_of_Cells_MMesh
```

I.1.21 Get Value Multi_Mesh Functions

The main documentation of the Get Value Multi_Mesh Functions in § 14.1.21 on page 182 contains additional explanation of this code listing.

```
define([POINTER_ACCESS_ROUTINE],[
  pushdef([VALUE], [$1])
  pushdef([VALUE_Result], expand(VALUE_Result))
  pushdef([Get_POINTER_VALUE_Multi_Mesh], expand(Get_VALUE_Multi_Mesh))

  function Get_POINTER_VALUE_Multi_Mesh (Mesh) result(VALUE_Result)

    ! Input variable.

    type(Multi_Mesh_type), target, intent(inout) :: Mesh ! Mesh object.

    ! Input/Output variable.

    ! Pointer to requested Base Structure.
    type(Base_Structure_type), pointer :: VALUE_Result

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)          ! Mesh is valid.

    ! Assign the pointer.

    VALUE_Result => Mesh%VALUE

    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)          ! Mesh is valid.

    return
  end function Get_POINTER_VALUE_Multi_Mesh

  popdef([VALUE])
  popdef([VALUE_Result])
  popdef([Get_POINTER_VALUE_Multi_Mesh])
])

fortext([Value],
        [Cell_Structure Node_Structure Face_Structure],[
  POINTER_ACCESS_ROUTINE(Value)
])

define([ACCESS_ROUTINE],[
  pushdef([VALUE], [$1])
```

```
  ifelse([$2], [2],
    [pushdef([DIMENSION], [, dimension(2)])],
    [pushdef([DIMENSION], [])])
  pushdef([Get_VALUE_Multi_Mesh], expand(Get_VALUE_Multi_Mesh))

  function Get_VALUE_Multi_Mesh (Mesh) result(VALUE)

    ! Input/Output variables.

    ! Base_Mesh object.
    type(Multi_Mesh_type), intent(in) :: Mesh

    ! Output variables.

    type(integer) DIMENSION :: VALUE       ! Multi_Mesh value to be output.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)     ! Mesh is valid.

    ! Set value.

    VALUE = Mesh%VALUE

    ! Verify guarantees - none.

    return
  end function Get_VALUE_Multi_Mesh

  popdef([VALUE])
  popdef([DIMENSION])
  popdef([Get_VALUE_Multi_Mesh])
])

fortext([Value],
        [First_Cell_PE Last_Cell_PE NCells_PE NCells_Total dnl
         First_Node_PE Last_Node_PE NNodes_PE NNodes_Total dnl
         First_Face_PE Last_Face_PE NFaces_PE NFaces_Total dnl
         Faces_per_Cell NDimensions],[
  ACCESS_ROUTINE(Value)
])

fortext([Value],
        [Range_Cells_PE Range_Nodes_PE Range_Faces_PE],[
  ACCESS_ROUTINE(Value, 2)
])

function Get_Name_Multi_Mesh (Mesh) result(Name)

  ! Input variable.

  type(Multi_Mesh_type), intent(in) :: Mesh  ! Variable to be queried.
```

```
    ! Output variable.

    type(character,name_length) :: Name             ! Name of Mesh.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)      ! Mesh is valid.

    ! Set the value.

    Name = Mesh%Name

    ! Verify guarantees - none.

    return
  end function Get_Name_Multi_Mesh
```

I.1.22 Get_Version_Multi_Mesh Procedure

The main documentation of the Get_Version_Multi_Mesh Procedure in § 14.1.22 on page 183 contains additional explanation of this code listing.

```
  function Get_Version_Multi_Mesh (Mesh) result(Version)

    ! Input variable.

    type(Multi_Mesh_type), intent(in) :: Mesh   ! Variable to be queried.

    ! Output variable.

    type(integer) :: Version                       ! Version number.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)      ! Mesh is valid.

    ! Get the value.

    Version = Mesh%Version

    ! Verify guarantees - none.

    return
  end function Get_Version_Multi_Mesh
```

I.1.23 Get_Volume_Cells_Multi_Mesh Procedure

The main documentation of the Get_Volume_Cells_Multi_Mesh Procedure in § 14.1.23 on page 183 contains additional explanation of this code listing.

```
  subroutine Get_Volume_Cells_Multi_Mesh (Volume_Cells, Mesh)

    ! Input variable.

    type(Multi_Mesh_type), intent(inout) :: Mesh  ! Mesh object.

    ! Input/Output variable.

    type(real,1) :: Volume_Cells ! Volume_Cells BNV.

    ! Internal variable.

    ! BNV of coordinates of nodes of cells.
    type(real,3) :: Coordinates_Nodes_of_Cells

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)                        ! Mesh is valid.
    VERIFY(Valid_State(Volume_Cells),5)                ! Volume_Cells is valid.
    VERIFY(Mesh%NCells_PE == SIZE(Volume_Cells),5)     ! Correct dimensions.

    ! Get Coordinates of the Nodes if needed.

    if (Mesh%Uniformity == "Nonuniform") then

      call Initialize (Coordinates_Nodes_of_Cells, Mesh%NDimensions, &
                       Mesh%NCells_PE, Mesh%Nodes_per_Cell)
      call Get_Coordinates_Nodes_of_Cells (Coordinates_Nodes_of_Cells, Mesh)

    end if

    ! Set the cell volumes.

    if (Mesh%Uniformity == "Uniform") then

      ! For a uniform mesh, all volumes are the same.
      Volume_Cells = Mesh%Volume_All_Cells

    else if (Mesh%Orthogonality == "Orthogonal") then

      ! For an orthogonal mesh, the volume may be calculated from the
      ! product of 1, 2 or 3 orthogonal edge lengths.
      select case (Mesh%NDimensions)
      case (1)
        ! Suppressed Y, Z.
        Volume_Cells = ( Coordinates_Nodes_of_Cells(1,:,2) &     ! Delta X -
```

```
                     - Coordinates_Nodes_of_Cells(1,:,1) )     ! Nodes 1 & 2
      case (2)
        ! Suppressed Z.
        Volume_Cells = ( Coordinates_Nodes_of_Cells(1,:,2) &     ! Delta X -
                     - Coordinates_Nodes_of_Cells(1,:,1) ) * & ! Nodes 1 & 2
                     ( Coordinates_Nodes_of_Cells(2,:,3) &     ! Delta Y -
                     - Coordinates_Nodes_of_Cells(2,:,1) )     ! Nodes 1 & 3
      case (3)
        Volume_Cells = ( Coordinates_Nodes_of_Cells(1,:,2) &     ! Delta X -
                     - Coordinates_Nodes_of_Cells(1,:,1) ) * & ! Nodes 1 & 2
                     ( Coordinates_Nodes_of_Cells(2,:,3) &     ! Delta Y -
                     - Coordinates_Nodes_of_Cells(2,:,1) ) * & ! Nodes 1 & 3
                     ( Coordinates_Nodes_of_Cells(3,:,5) &     ! Delta Z -
                     - Coordinates_Nodes_of_Cells(3,:,1) )     ! Nodes 1 & 5
      end select

    else
      ! Coding for other mesh types not implemented yet.
      VERIFY(.false.,1)
    end if

    ! Finalize temporaries.

    if (Mesh%Uniformity == "Nonuniform") then
      call Finalize (Coordinates_Nodes_of_Cells)
    end if

    ! Verify guarantees.

    VERIFY(Valid_State(Mesh),5)          ! Mesh is valid.
    VERIFY(Valid_State(Volume_Cells),5)  ! Volume_Cells still valid.

    return
  end subroutine Get_Volume_Cells_Multi_Mesh
```

I.1.24 Set_Coordinates_Multi_Mesh Procedure

The main documentation of the Set_Coordinates_Multi_Mesh Procedure in § 14.1.24 on page 183 contains additional explanation of this code listing.

```
!  define([SET_COORDINATES_ROUTINE],[
!    pushdef([DIM], [$1])
!    pushdef([Set_Coordinates_Multi_Mesh_DIM],
!      expand(Set_Coordinates_Multi_Mesh_DIM))

!    subroutine Set_Coordinates_Multi_Mesh_DIM (Mesh, Coordinates)

!      ! Input variable.

!      type(real,DIM,np), intent(in) :: Coordinates       ! Coordinates bare naked array.

!      ! Input/Output variable.
```

```
!      type(Multi_Mesh_type), intent(inout) :: Mesh ! Variable to be set.

!      !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

!      ! Verify requirements.

!      VERIFY(Valid_State(Mesh),5)                      ! Mesh is valid.
!      VERIFY(Valid_State_NP(Coordinates),5)               ! Coordinates is valid.
!      VERIFY($1 == Mesh%A_Dimensionality,5)   ! Mesh has been set up for this call.
!      VERIFY(SHAPE(Coordinates) == SHAPE(Mesh%Coordinates$1),5)  ! Coordinates shape check.

!      ! Set the coordinates.

!      Mesh%Coordinates$1 = Coordinates

!      ! Increment the version number.

!      Mesh%Version = Mesh%Version + Version_Increment

!      ! Verify guarantees.

!      VERIFY(Valid_State(Mesh),5)                            ! Mesh is still valid.

!      return
!    end subroutine Set_Coordinates_Multi_Mesh_DIM

!    popdef([DIM])
!    popdef([Set_Coordinates_Multi_Mesh_DIM])
!  ])

!  forloop([Dim],[1],[5],[
!    SET_COORDINATES_ROUTINE(Dim)
!  ])
```

I.1.25 Set_Version_Multi_Mesh Procedure

The main documentation of the Set_Version_Multi_Mesh Procedure in § 14.1.25 on page 184 contains additional explanation of this code listing.

```
  subroutine Set_Version_Multi_Mesh (Mesh, Version)

    ! Input variable.

    type(integer), intent(in) :: Version             ! Version number.

    ! Input/Output variable.

    type(Multi_Mesh_type), intent(inout) :: Mesh  ! Variable to be set.

    !~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    ! Verify requirements.

    VERIFY(Valid_State(Mesh),5)      ! Mesh is valid.

    ! Set the value.

    Mesh%Version = Version

    ! Verify guarantees - none.

    return
  end subroutine Set_Version_Multi_Mesh
```

I.1.26 Multi_Mesh Class Unit Test Program

This lightly commented program performs a unit test on the Multi_Mesh Class, which is described in § 14.1 on page 169.

```
program Unit_Test
  use Caesar_Data_Structures_Module
  use Caesar_Mathematics_Module
  use Caesar_Multi_Mesh_Class
  use Caesar_Numbers_Module, only: one, four, zero
  implicit none

  type(Communication_type) :: Comm
  type(Base_Structure_type) :: Cell_Structure, Node_Structure
  type(integer,2) :: Nodes_of_Cells_Index_Values
  type(Data_Index_type) :: Nodes_of_Cells_Index
  type(Assembled_Vector_type) :: Coordinates_Nodes_AV
  type(Distributed_Vector_type) :: Coordinates_Nodes_DV, Results_Cells_DV
  type(Overlapped_Vector_type) :: Coordinates_Nodes_of_Cells_OV
  type(Collected_Array_type) :: Coordinates_Nodes_of_Cells_CA
  type(Status_type) :: status
  type(character,name_length) :: Name_Name
  type(real,2) :: Coordinates, Results_Cells_BNV
  type(real,3) :: Processed_Coordinates
  type(integer) :: c, NodeSize, d, Dimensionality, DimSize, i, j, &
                   Many_Axis_Length, n, NDimensions, Nodes_per_Cell, &
                   NNodes
  type(character,80,1) :: Output_Buffer
  type(logical) :: detailed_output, Success

  ! Initializations.

  call Initialize (Comm)
  call Output (Comm)
  call Initialize (status)
  call Initialize (Name_Name)
  Dimensionality = 2
  NDimensions = 2
  detailed_output = NPEs <= 8
```

```
! Set up the Shell Partition Structures.

call Initialize_Shell_Partition (NDimensions, Cell_Structure, &
                                 Node_Structure, Nodes_of_Cells_Index, &
                                 detailed_output)

! Initialize AV, DV, OV and CA Coordinate vectors.

Name_Name = 'Coordinates of Nodes'
call Initialize (Coordinates_Nodes_AV, Node_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)
call Initialize (Coordinates_Nodes_DV, Node_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)
Name_Name = 'Coordinates of Nodes of Cells'
call Initialize (Coordinates_Nodes_of_Cells_OV, Nodes_of_Cells_Index, &
                 Dimensionality, Name_Name, status, NDimensions)
call Initialize (Coordinates_Nodes_of_Cells_CA, &
                 Coordinates_Nodes_of_Cells_OV, Name_Name, status)
Name_Name = 'Results of Cells'
call Initialize (Results_Cells_DV, Cell_Structure, Dimensionality, &
                 Name_Name, status, NDimensions)

! Set up Coordinates array on IO PE only.

NNodes = Node_Structure%Length_Total
if (this_is_IO_PE) then
  DimSize = NDimensions
  NodeSize = NNodes
else
  DimSize = 0
  NodeSize = 0
end if
call Initialize (Coordinates, DimSize, NodeSize)
if (this_is_IO_PE) then
  Coordinates(1,:) = (/ ( changetype(real,(n)), n = 1,NNodes ) /)
  Coordinates(2,:) = (/ ( one, n = 1,NNodes ) /)
end if

Name_Name = ''

! Set up Processed Coordinates array on every processor.

Nodes_per_Cell = 2**NDimensions
Many_Axis_Length = Nodes_per_Cell
call Initialize (Processed_Coordinates, NDimensions, &
                 Cell_Structure%Length_PE, Many_Axis_Length)

! Set up Results_Cells_BNV array on every processor.

Nodes_per_Cell = 2**NDimensions
call Initialize (Results_Cells_BNV, NDimensions, &
                 Cell_Structure%Length_PE)
```

```
! Version number check.

Coordinates_Nodes_of_Cells_CA = 123
Success = Version(Coordinates_Nodes_of_Cells_CA) == 123
call Output_Test ('Version number', Success)

! Send Coordinates into Assembled Vector, then Distributed Vector,
! then Collected Array, and access the data.

Coordinates_Nodes_AV = Coordinates
Coordinates_Nodes_DV = Coordinates_Nodes_AV
Coordinates_Nodes_of_Cells_CA = Coordinates_Nodes_DV
Processed_Coordinates = Coordinates_Nodes_of_Cells_CA

! Re-construct the original Nodes_of_Cells index values for comparison.

call Initialize (Nodes_of_Cells_Index_Values, &
                 SIZE(Nodes_of_Cells_Index%Index2,1), &
                 SIZE(Nodes_of_Cells_Index%Index2,2))
Nodes_of_Cells_Index_Values = Nodes_of_Cells_Index

! Check to see if the Processed Coordinates are correct.

Success = Global_ALL(INT(Processed_Coordinates(1,:,:)) == &
          Nodes_of_Cells_Index_Values(:,:))
call Output_Test ('BNV-AV-DV-CA-BNA Index', Success)

Success = Global_ALL(Processed_Coordinates(2,:,:) == one)
call Output_Test ('BNV-AV-DV-CA-BNA One', Success)

! A different cycle that includes an Overlapped Vector.

Processed_Coordinates = zero
Coordinates_Nodes_AV = Coordinates
Coordinates_Nodes_DV = Coordinates_Nodes_AV
Coordinates_Nodes_of_Cells_OV = Coordinates_Nodes_DV
Coordinates_Nodes_of_Cells_CA = Coordinates_Nodes_of_Cells_OV
Processed_Coordinates = Coordinates_Nodes_of_Cells_CA

! Check to see if the Processed Coordinates are correct.

Success = Global_ALL(INT(Processed_Coordinates(1,:,:)) == &
          Nodes_of_Cells_Index_Values(:,:))
call Output_Test ('BNV-AV-DV-OV-CA-BNA Index', Success)

Success = Global_ALL(Processed_Coordinates(2,:,:) == one)
call Output_Test ('BNV-AV-DV-OV-CA-BNA One', Success)

! Combination tests:

!   Average test.

call Combine_with_Average (Results_Cells_DV, Coordinates_Nodes_of_Cells_CA)
Results_Cells_BNV = Results_Cells_DV
```

```
Success = Global_ALL(Results_Cells_BNV(1,:) == &
          changetype(real, SUM(Nodes_of_Cells_Index_Values(:,:),2)) /  &
          changetype(real, Nodes_per_Cell))
call Output_Test ('Average index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Average one', Success)

!   Max test.

call Combine_with_MAX (Results_Cells_DV, Coordinates_Nodes_of_Cells_CA)
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          MAXVAL(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Max index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Max one', Success)

!   Min test.

call Combine_with_MIN (Results_Cells_DV, Coordinates_Nodes_of_Cells_CA)
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          MINVAL(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Min index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == one)
call Output_Test ('Min one', Success)

!   Sum test.

Results_Cells_DV = Coordinates_Nodes_of_Cells_CA  ! Sum is the default.
Results_Cells_BNV = Results_Cells_DV

Success = Global_ALL(INT(Results_Cells_BNV(1,:)) == &
          SUM(Nodes_of_Cells_Index_Values(:,:),2))
call Output_Test ('Sum index', Success)
Success = Global_ALL(Results_Cells_BNV(2,:) == four)
call Output_Test ('Sum four', Success)

! Output statements.

call Output (Coordinates_Nodes_of_Cells_CA, &
  MAX(1, Cell_Structure%Length_Total/10), &
  MIN(Cell_Structure%Length_Total, Cell_Structure%Length_Total/10+50) &
)

! Check state of various objects.

VERIFY(Valid_State(Coordinates_Nodes_AV),0)
VERIFY(Valid_State(Coordinates_Nodes_DV),0)
VERIFY(Valid_State(Coordinates_Nodes_of_Cells_CA),0)
VERIFY(Valid_State(Cell_Structure),0)
```

```
  VERIFY(Valid_State(Node_Structure),0)
  VERIFY(Valid_State(Nodes_of_Cells_Index),0)

  ! Finalize data structures and communications.

  call Finalize (Results_Cells_DV)
  call Finalize (Results_Cells_BNV)
  call Finalize (Nodes_of_Cells_Index_Values)
  call Finalize (Coordinates_Nodes_of_Cells_CA)
  call Finalize (Coordinates_Nodes_AV)
  call Finalize (Coordinates_Nodes_DV)
  call Finalize (Coordinates)
  call Finalize (Processed_Coordinates)
  call Finalize (Nodes_of_Cells_Index)
  call Finalize (Cell_Structure)
  call Finalize (Node_Structure)
  call Finalize (Comm)

end
```

Bibliography

CM Fortran Reference Manual (1989), Cambridge, MA. Version 5.2-0.6.

Golub, Gene H. and Charles F. Van Loan (1989), *Matrix Computations*, The Johns Hopkins University Press. Second Edition, ISBN 0-8018-3739-1.

Knuth, Donald E. (1992), *Literate Programming*, (Stanford, California: Center for the Study of Language and Information). CSLI Lecture Notes, No. 27, ISBN 0-937073-80-6. Online information is available at `http://www-cs-faculty.stanford.edu/~knuth/lp.html`.

Lakos, John (1996), *Large-Scale C++ Software Design*, Addison Wesley. ISBN 0-201-63362-0.

Meyer, Bertrand (1997), *Object-Oriented Software Construction*, second edn, ISE Inc. ISBN 0-13-629155-4. Online information is available at `http://archive.eiffel.com/doc/oosc/`.

Index