Earth System Modeling Framework
Software Developer's Guide

Silverio Vasquez, Sylvia Murphy, Cecelia DeLuca, Walter Spector, Gerhard Theurich

August 26, 2020

---

http://www.earthsystemmodeling.org

Contents

1 Introduction **5**
- 1.1 About this Document . . . 5
- 1.2 Supplementary Information . . . 5
- 1.3 Acknowledgments . . . 5

2 Groups and Roles in ESMF Development **5**
- 2.1 Core Team . . . 5
  - 2.1.1 Core Team Roles . . . 5
- 2.2 Joint Specification Team (JST) . . . 7
- 2.3 Change Review Board (CRB) . . . 7
- 2.4 Quality Assurance Responsibilities . . . 7

3 Collaboration Environment and Communication **7**
- 3.1 Mailing Lists . . . 7
- 3.2 Meetings and Telecons . . . 8
- 3.3 SourceForge Open Source Development Environment . . . 8
  - 3.3.1 The Main ESMF Site and Repository . . . 8
  - 3.3.2 The ESMF Contributions Site and Repository . . . 9

4 Processes **9**
- 4.1 Software Process Model . . . 9
- 4.2 ESMF Process History . . . 9
  - 4.2.1 Software Concept . . . 9
  - 4.2.2 Requirements Analysis . . . 10
  - 4.2.3 Architectural Design . . . 10
- 4.3 Ongoing Development . . . 10
  - 4.3.1 Telecon Etiquette . . . 10
  - 4.3.2 Design Reviews . . . 10
  - 4.3.3 Implementation and Test Before Internal Release . . . 11
  - 4.3.4 Implementation and Test Before Public Release . . . 11
  - 4.3.5 Code Check-In . . . 12
  - 4.3.6 Code Reviews . . . 12
  - 4.3.7 Releases . . . 12
  - 4.3.8 Backups . . . 13
- 4.4 Testing and Validation . . . 13
  - 4.4.1 Unit Tests . . . 14
  - 4.4.2 Examples . . . 18
  - 4.4.3 System Tests . . . 19
  - 4.4.4 Test Harness . . . 20
  - 4.4.5 Use Test Cases (UTCs) . . . 21
  - 4.4.6 MAPL Testing . . . 21
  - 4.4.7 Beta Testing . . . 22
  - 4.4.8 Automated Regression Tests . . . 22
  - 4.4.9 Investigating Test Failures . . . 22
  - 4.4.10 Building the Documentation . . . 23
  - 4.4.11 Testing for Releases . . . 24
- 4.5 User Support . . . 24
  - 4.5.1 Roles . . . 24
  - 4.5.2 Support Categories . . . 24

4.5.3 Summary Work Flow . . . 24
4.5.4 General Guidelines for Handling Tickets . . . 25
4.5.5 esmf_support@ucar.edu Mail Archives . . . 25
4.5.6 INFO:Code (subject) mail messages . . . 26
4.5.7 freeCRM . . . 26
4.5.8 Annual Code Contact . . . 27
4.5.9 Dealing with Applications that use ESMF . . . 27

5 Apps **28**

6 Test Harness **28**
6.1 Specifying Test Harness Tests . . . 28
6.2 Running Test Harness Tests as Part of Unit Tests . . . 29
6.3 Invoking a Single Test Harness Test Case Using gmake . . . 29
6.4 Invoking a Single Test Harness Test Case from the Command Line . . . 30
6.5 Top Level Configuration File . . . 30
6.5.1 Problem descriptor file . . . 31
6.5.2 Problem descriptor string syntax . . . 32
6.5.3 General Data Structures . . . 35
6.5.4 Specifier files . . . 35
6.5.5 Grid Specification . . . 36
6.5.6 Distribution Specification . . . 41
6.5.7 Class Specification . . . 42
6.6 Reporting test results . . . 43

7 Conventions **43**
7.1 Docs: Code and Documentation Templates and Associated Scripts . . . 44
7.1.1 Documentation Generation Script . . . 44
7.1.2 Code Generation Scripts . . . 44
7.2 Docs: Documentation Guidelines and Conventions . . . 44
7.2.1 Accessibility . . . 44
7.2.2 File format . . . 44
7.2.3 Typeface and Diagram Conventions . . . 45
7.2.4 Style Rules for LaTeX . . . 45
7.3 Docs: Performance Report Conventions . . . 45
7.4 Docs: Reference Manual Conventions . . . 46
7.4.1 Description, Use and Examples, and Other Introductory Sections . . . 46
7.4.2 Examples Sections . . . 46
7.4.3 Flags and Options Sections . . . 46
7.4.4 Class API Sections . . . 46
7.5 Code: Method Conventions . . . 47
7.5.1 Standard Method Names . . . 47
7.5.2 Use of *Set and *Get . . . 48
7.5.3 Use of Is* and Has* . . . 48
7.5.4 Functions vs. Subroutines . . . 48
7.5.5 Source and Destination Ordering . . . 48
7.6 Code: Argument Conventions . . . 48
7.6.1 Standard Variable Names . . . 48
7.6.2 Use of Is* and Has* . . . 49
7.6.3 Variable Capitalization . . . 49
7.6.4 Variables Associated with Options . . . 49

7.6.5 Variables Having Logical Data Type . . . 49
7.6.6 Arguments which are Arrays . . . 49
7.6.7 Arguments which are Pointers . . . 49
7.7 Code: File Rules . . . 49
7.7.1 Version Identification . . . 49
7.7.2 License and Copyright Information . . . 50
7.7.3 TODO: Reminder . . . 50
7.7.4 FILENAME Macros . . . 50
7.8 Code: Style Rules for Source Code . . . 50
7.9 Code: Error Handling Conventions . . . 51
7.9.1 Objectives . . . 51
7.9.2 Approach . . . 51
7.9.3 Error Masking . . . 52
7.9.4 Example (pre-review method) . . . 52
7.9.5 Example (post-review method) . . . 53
7.9.6 Memory Allocation Checking . . . 54
7.10 Initialization Standardization Instructions . . . 55
7.10.1 Overview . . . 55
7.10.2 Instructions . . . 56
7.10.3 Module . . . 56
7.10.4 Shallow Class . . . 56
7.10.5 Deep Class . . . 60
7.10.6 Parameter Class . . . 63
7.10.7 Subroutine . . . 63
7.10.8 ESMF Class Types . . . 64
7.11 Code: Data Type Consistency Guidelines . . . 64
7.11.1 Use ESMF names for data types in C and ESMF data kinds in Fortran . . . 65
7.11.2 Fortran . . . 65
7.11.3 C and C++ . . . 67
7.11.4 ESMF_TypeKind_Flag “labels”: When knowing the data type/kind is necessary . . . 68
7.11.5 Guidelines for Fortran-C Interfaces . . . 69
7.12 Code: Optional Argument Conventions for the C/C++ API . . . 70
7.12.1 Overview . . . 70
7.12.2 Approach . . . 71
7.12.3 Internal Macros for Processing the Optional Argument List . . . 72
7.12.4 Parsing the Optional Argument List . . . 73
7.13 Code: Makefile Conventions . . . 75
7.13.1 Code Building Rules . . . 75
7.13.2 Document Building Rules . . . 75
7.13.3 Include Files . . . 76
7.14 Preprocessor Usage . . . 76
7.14.1 Using the Preprocessor For Generic Fortran Code . . . 76
7.14.2 System Dependent Strategy Using Preprocessor . . . 78
7.15 ESMF Data Type Autopromotion Support Policy and Guide . . . 78
7.15.1 How We Arrived at This Autopromotion Support Policy . . . 80
7.16 Scripts: Script Coding Standard . . . 82
7.16.1 Content Rules . . . 82
7.17 Lang: Interlanguage Coding Conventions . . . 82
7.17.1 Optional Arguments Across Language Interfaces . . . 82
7.18 Lang: Fortran Coding Standard . . . 83
7.18.1 Content Rules . . . 83

7.18.2 Style Rules . . . 86
7.19 Lang: C/C++ Coding Standard . . . 87
7.20 Repo: Source Code Naming and Tagging Conventions . . . 88
7.20.1 Public Releases . . . 88
7.20.2 Internal Releases . . . 88
7.21 Data Management Conventions . . . 89

8 Tracking and Metrics **89**
8.1 Release Schedule . . . 89
8.2 Task List . . . 89
8.3 Trackers . . . 89
8.3.1 Setting Ticket Priorities . . . 89
8.3.2 Labeling Tickets Longer than 2 Weeks . . . 89
8.3.3 Estimating Ticket Completion Time . . . 90
8.3.4 Labeling Tickets with Time Estimates . . . 90
8.3.5 Cross-Referencing the Task List and Trackers . . . 90
8.3.6 Summary . . . 90
8.3.7 Types of Trackers . . . 91
8.4 Metrics . . . 91
8.4.1 Unit and System Tests Coverage . . . 91
8.4.2 ESMF Requirements Coverage . . . 91
8.4.3 Source Lines of Code, SLOC . . . 92

9 Policies **92**
9.1 External Software Libraries . . . 92
9.2 Graphics Packages . . . 92

Appendix A: Testing Terminology **93**

Bibliography **94**

1 Introduction

1.1 About this Document

The Earth System Modeling Framework (ESMF) *Software Developer's Guide*, or simply the *Guide*, is the reference handbook that describes the practices, standards, and conventions recommended for ESMF core software development. It is updated as needed.

Suggestions on how to improve the *Guide* should be sent to esmf_support@ucar.edu.

1.2 Supplementary Information

An important source of supplementary information is the ESMF website:

http://www.earthsystemmodeling.org

This website is referred to throughout the *Guide*. It includes a link to the ESMF CVS repository, the current release schedule, task lists, and many other types of project information.

The *Guide* doesn't contain instructions on how to build ESMF, adapt user codes for ESMF, or use the ESMF interfaces. For documentation corresponding to the last public release, see the **Users** tab on the ESMF website. For documentation of all releases, including internal releases and previous public releases, click the **Download** tab on the website and then **View All Releases**.

1.3 Acknowledgments

Sections of this document were derived from the *GFDL Flexible Modeling System Developers' Manual* [2] and the now-defunct *Community Climate System Model Software Developer's Guide* [7]. These have been adapted with the permission of the respective authors.

2 Groups and Roles in ESMF Development

For more detail on the groups below, including their Terms of Reference, see the *ESMF Project Plan*[5]. This document is available via the **Management** link on the ESMF website navigation bar.

2.1 Core Team

ESMF software implementation is led by a distributed **Core Team** which is based in the Technology Development Division (TDD) of the National Center for Atmospheric Research (NCAR). The Core Team relies on close interaction with customers, and the work of many contributors. Core Team activities are open to all active ESMF developers.

All Core Team members are responsible for helping to develop effective project processes and for following processes that are in place.

2.1.1 Core Team Roles

Developers are the Core Team members that design, implement, document, and test ESMF software. They are expected to interact with customers throughout the entire development process in order to understand customer requirements. Developer priorities are set by the Change Review Board (CRB) but developers are expected to understand customer requirements and needs and to manage their own time.

The **Integrator** is the lead tester. He or she is responsible for running regression tests, managing project computing accounts, preparing for releases, generating project metrics, and overseeing request and bug tracking.

The **Core Team Manager** is responsible for overseeing the overall development, and for coordinating the activities of multiple developers so that project schedules and priorities are achieved. Responsibilities include:

- Project administration and the assignment of tasks
- Acquisition of funding
- Deciding on short-term priorities based on longer-term objectives
- Monitoring conformance to processes and conventions
- Representation of the Core Team to NCAR executive management, the CRB, and sponsors.

Code **Advocates** are Core Team members who have been assigned to interface with owners of a particular code e.g. CCSM, GEOS-5. Advocates are expected to contact the code owners and keep track of what and how they are doing. They should know the following things about their assigned code:

- contact info
- models or components being ESMF-ized
- what the customer is trying to do
- what platforms they are working on
- what pieces of ESMF they are using
- their current status of ESMF-ization
- what is holding them up if anything
- open tickets (see section 8.3 for types) and their status

Code **Advocates** should do the following for their code:

- Touch base periodically and find out how things are going
- Coordinate the resolution of all tickets (see section 8.3 for types)
- Be prepared to brief the status of all requests and issues
- Update or ensure the update of related tickets (see Section 4.5 for details on the proper handling of tickets).

Advocates are NOT responsible for:

- Representing the code to the CRB
- Fielding support related phone calls.

  All customers should send support requests to esmf_support@ucar.edu. They should not call their Advocate or other developers. Dire emergencies are exempt from the no-phone rule. There are many good reason for doing this, including developer absences, group visibility and communication, and enabling developers to set priorities. If a customer really feels the need to talk with an Advocate or developer, they need to arrange for a date and time through the support list.

Handlers are developers assigned to fix a particular problem reported on one of the trackers (see 8.3 for list). They are expected to be sufficiently familiar with the issue and the code involved so as to not unnecessarily pester the customer with requests for information (e.g. the information is contained in previous tickets from the same customer). See section 4.5 for a complete list of **Handlers** activities in the support process.

Support Lead is a Core Team member who has been assigned the responsibility for monitoring customer relations and the quality of the customer support process. Specific duties include:

- Ensure IMAP folders are properly named and utilized

- Assist Advocates in understanding the status of support requests related to their codes
- Clear out mail list spam filters and clear all pending moderation requests
- Conduct monthly support request review meetings
- With the assistance of the Advocates, update the Core Team on the status of codes
- Monitor the teams customer relations management (CRM) software and database

2.2 Joint Specification Team (JST)

The Core Team, along with contributors, customers, technical managers, and other stakeholders in development, are collectively referred to as the **ESMF Joint Specification Team**, or **JST**. JST membership is open to members of the science, computing, and related communities.

2.3 Change Review Board (CRB)

ESMF development priorities and schedules are set by a **Change Review Board** (CRB) that consists of individuals associated with major ESMF initiatives and applications. Members of the CRB are chosen by the ESMF Executive Management.

2.4 Quality Assurance Responsibilities

Collective public reviews and the intelligence and attentiveness of project staff are the two primary mechanisms for software quality assurance.

Requirements, design, code, and project documents such as plans are subject to public review. The purpose of the reviews is to look for models of good documentation, as well as inconsistencies, errors, inefficiencies, and areas for improvement. Reviews also increase coordination and awareness within the ESMF project.

The Integrator ensures that support requests and bugs are reported, tracked, and resolved, and that automated test and backup scripts are operating correctly.

The Core Team Manager works with team to ensure that documentation is sufficient, tracks consistency between documentation and source code before releases, and monitors conformance to coding and documentation standards. The Core Team Manager is also responsible for ensuring the quality and accuracy of the ESMF source code distribution overall, by leading the development, implementation and documentation of development, test, and release procedures.

3 Collaboration Environment and Communication

3.1 Mailing Lists

The main ESMF technical mailing list is:

esmf_jst@cgd.ucar.edu

This list handles telecon announcements, technical questions and comments, discussions and materials relating to design and code reviews, planning discussions, project announcements, and any other items that concern the JST.

The list for active ESMF developers is:

esmf_core@cgd.ucar.edu

This list is for people who are developing code and checking it into the ESMF repository. Mails here may cover coordination issues relating to check-ins and releases, technical material that is not yet ready for the JST, and information specific to NCAR and particular development platforms.

Support questions should be directed to:

esmf_support@ucar.edu

The **Users** tab on the ESMF website describes how to submit an effective support request and outlines the ESMF support policy.

People who are interested in occasional high-level project updates can join the mailing list:

esmf_info@cgd.ucar.edu

Subscribers to this list receive more or less quarterly newletters describing ESMF achievements and events.

To subscribe to any of these lists on-line, see the **Users** tab on the ESMF website.

3.2 Meetings and Telecons

ESMF JST telecons are held as needed, typically on a weekly basis. Normal telecon times are 1:00pm MT Thursdays, and sometimes Tuesdays. A calendar of telecon topics is maintained on the home page of the ESMF website. These telecons are open and are announced to the esmf_jst@cgd.ucar.edu list.

The Core Team meets weekly, at 9:30am MT on Wednesdays. The meeting is also set up as a telecon. Core Team meetings are open to active developers.

The ESMF project has open community meetings on an annual basis, usually during late spring.

CRB meetings are held quarterly and are closed. However, prior to each CRB meeting a JST telecon is devoted to collecting comments, requirements, and priorities from JST members for consideration by the CRB.

3.3 SourceForge Open Source Development Environment

The ESMF project utilizes the SourceForge open source development environment. It provides a variety of tools as well as web-browsable repositories.

3.3.1 The Main ESMF Site and Repository

The main ESMF SourceForge site is at:

http://sourceforge.net/projects/esmf

This site is accessible from the ESMF website, via either the **Developers** link or the *SourceForge* logo on the navigation bar. It is used for

- hosting and browsing the ESMF source code CVS repository;
- maintaining task lists;
- archiving mailing lists;
- providing tarballs of public source code releases (releases are also made available on the ESMF website);
- tracking bugs; and
- tracking support requests.

The source code and tools on this site are maintained by the ESMF Core Team, and contributions and changes cannot be made to them without coordination with the Core Team.

The main ESMF CVS repository is web-accessible at: **http://sourceforge.net/cvs/?group_id=38089**

The SourceForge site has instructions for checking out code. Other CVS documentation is available at: **http://www.gnu.org/manual/cvs/html_node/cvs_toc.html**

All ESMF documents, source code and test and other scripts are stored in the main repository.

The SourceForge repository contains only the ESMF framework. Components that use ESMF are stored in repositories at their home institutions.

3.3.2 The ESMF Contributions Site and Repository

A second ESMF SourceForge site and repository is at:

http://sourceforge.net/projects/esmfcontrib

Contributors in the broader community can use this site to archive and share code related to ESMF. Coordination is not required with the ESMF Core Team in order to check into the contributions repository.

4 Processes

The ESMF development environment has several defining characteristics. First, both the ESMF Core Team and the JST are distributed. This makes incorporating simple, efficient communication mechanisms into the development process essential. Second, the JST and Core Team work on a range of different platforms, at sites that don't have the time, resources, or inclination to install demanding packages. Collaboration tools that require no purchase or installation before use are essential. Finally, ESMF is committed to open development. As much as possible, the ESMF team tries to keep the workings of the project - metrics, support and bug lists, schedules, task lists, source code, you name it - visible to the broad community.

4.1 Software Process Model

The ESMF software development cycle is based on the staged delivery model [8]. The steps in this software development model are:

1. **Software Concept** Collect and itemize the high-level requirements of the system and identify the basic functions that the system must perform.
2. **Requirements Analysis** Write and review a requirements document - a detailed statement of the scientific and computational requirements for the software.
3. **Architectural Design** Define a high-level software architecture that outlines the functions, relationships, and interfaces for major components. Write and review an architecture document.
4. **Stage 1, 2, ..., n** Repeat the following steps creating a potentially releasable product at the end of each stage. Each stage produces a more robust, complete version of the software.
   - *Detailed Design* Create a detailed design document and API specification. Incorporate the interface specification into a detailed design document and review the design.
   - *Code Construction and Unit Testing* Implement the interface, debug and unit test.
   - *System Testing* Assemble the complete system, verify that the code satisfies all requirements.
   - *Release* Create a potentially releasable product, including User's Guide and User's Reference Manual. Frequently code produced at intermediate stages software will be used internally.
5. **Code Distribution and Maintenance** Official public release of the software, beginning of maintenance phase.

We have customized and extended this standard model to suit the ESMF project. At this stage of ESMF development, we are in the iterative design/implement/release cycle. Below are a few notes on earlier stages.

4.2 ESMF Process History

4.2.1 Software Concept

Participants in the ESMF project completed the Software Concept stage in the process of developing a unified set of proposals. A summary of the high-level requirements of ESMF - a statement of project scope and vision - is included in the *General Requirements* part of the *ESMF Requirements Document*[4]. This was a successful effort in defining the scope of the project and agreeing to an overall design strategy.

4.2.2 Requirements Analysis

The ESMF Team spent about six months at the start of the project producing the *ESMF Requirements Document*. This outlined the major ESMF capabilities necessary to meet project milestones and achieve project goals. The second part of the document was a detailed requirements specification for each functionality class included in the framework. This document also included a discussion of the process that was used to initially collect requirements. The *Requirements Document* was a useful reference for the development team, especially for new developers coming in from outside of the Earth science domain. However, as the framework matured, support requests and the Change Review Board process took precedence in defining development tasks and setting priorities. The *Requirements Document* is bundled with the ESMF source distribution through version 2; with version 3 it was removed.

4.2.3 Architectural Design

The project had difficulty with the *Architecture Document*. The comments received back on the completed work, informally and from a peer review body, indicated that the presentation of the document was ineffective at conveying how the ESMF worked. Although the document was full of detailed and complex diagrams, the terminology and diagrams were oriented to software engineers and were not especially scientist-friendly. The detailed diagrams also made the document difficult to maintain. This experience helped to guide the ESMF project towards more user-oriented documents, but it also left a gap in the documentation that has taken time to fill.

4.3 Ongoing Development

The following are processes the ESMF team is actively following. These guidelines apply to core team developers and outside contributors who will be checking code into the main ESMF repository.

All design and code reviews are telecons held with the JST. Telecons are scheduled with the Core Team Manager, put on the ESMF calendar on the home page of the ESMF website, and announced on the esmf_jst@cgd.ucar.edu list.

4.3.1 Telecon Etiquette

When you call in, it's nice to give your name at the first opportunity. Telecon hosts will make an effort to introduce people on the JST calls, especially first-timers. Please don't put the telecon on hold (we sometimes get telecon-stopping music or beeps this way).

Within a week or so after the telecon, the host (the developer if it's a design or code review) is expected to send out a summary to esmf_jst@cgd.ucar.edu with the date of the call, the participants, and notes or conclusions.

4.3.2 Design Reviews

1. Introductory telecon(s). The point here is to scope out the problem at hand. These calls cover the following, as they apply.

   - Understand the capability needed and review requirements.
   - Discuss design alternatives.
   - Survey and discuss any existing packages that cover the new functionality.
   - Discuss potential use test cases.
   - Figure out which customers will be involved in use test case development and identify customers that are likely to provide important input (sometimes we offer a friendly reminder service to these folks before relevant telecons).

   For these introductory discussions, any form of material is fine - diagrams, slides, plain text ramblings, lists of questions, ...

2. Initial design review(s). The document presented should be in the format of the ESMF Reference Manual, either in plain text or in latex/ProTeX. This is so the document can be incorporated into project documentation after implementation. The initial review document should include at least the following sections:

   - Class description.
   - Use and Examples. Here the examples begin with the very simplest cases.
   - API sufficient to cover the examples. This is because it can be difficult to follow the examples (e.g. tell what arguments and options are) without the basic API entries accompanying them.

   This step is iterated until developers and customers converge.

3. Full telecon review(s). The developer should prepare the API specification using latex and ProTex following the conventions in the Reference Manual. Most of the Reference Manual section(s) for the new or modified class(es), including Class Options and Restrictions and Future Work, should be available at the time of this review. Diagrams should be ready here too.

4. Use test case telecon review. For each major piece of functionality, a use test case is prepared in collaboration with customers and executed before release. The use test case is performed on a realistic number of processors and with realistic input data sets.

   It doesn't have to work (and probably won't) before it's reviewed, but it needs to work before the functionality appears in a release. The developer checks it into the top-level use_test_case directory on SourceForge and prepares a HTML page outlining it for the Test & Validation page on the ESMF website. Unlike unit and system tests, use test cases aren't distributed with the ESMF source.

4.3.3 Implementation and Test Before Internal Release

Code should be written in accordance with the interface specifications agreed to during design reviews and the coding conventions described in Section 7.

There is an internal release checklist on the Test & Validation page of the ESMF website that contains an exhaustive listing of develop and tester responsibilities for releases. For additional discussion of test and validation procedures, see Section 4.4.

The developer is responsible for working with the tester(s) to make sure that the following are present before an internal release:

- 100% unit and system test coverage of new interfaces, with the exception of interfaces where type/kind/rank is heavily overloaded. All arguments tested at least once.
- Use test cases work

4.3.4 Implementation and Test Before Public Release

There is a public release checklist on the Test & Validation page of the ESMF website that contains an exhaustive listing of develop and tester responsibilities for releases. For additional discussion of test and validation procedures, see Section 4.4.

Same as for internal release, plus:

- Design and Implementation Notes section for Reference Manual complete.
- Developer and tester ensure that test coverage for new interfaces is sufficient, implementing any additional tests to make it so. This includes testing of options and tests for error handling and recovery.

4.3.5 Code Check-In

Developers are encouraged to check their changes into the repository as they complete them, as frequently as possible without breaking the existing code base.

1. Both core and contributors should test on at least three compilers before commit.
2. For core team developers, a mail should go out to esmf_core@cgd.ucar.edu before check-in for very large commits and for commits that will break the HEAD. For contributors a mail should go out to esmf_core@cgd.ucar.edu before ANY commit.
3. No code commits should be made between 0.00 and 4:00 Mountain Time. During this time the regression tests scripts are checking out code and any commits will lead to inconsistent test results which are hard to interpret.
4. Core team developers can be set up to receive email from SourceForge for every check-in, by writing esmf_support@ucar.edu with the request.

To accomplish the first item on the list after a commit of source code, an email can be sent to esmftest@cgd,ucar.edu with the exact subject "Run_ESMF_Test_Build". The mailbox is checked every quarter hour on the quarter hour. This email initiates a test on pluto that builds and installs ESMF with four compilers: g95, gfortran, lahey, and nag, with ESMF_BOPT set to "g" and "O".

When the test is started an email with the subject "ESMF_Test_Builds_Pluto_started", is sent to esmf_core@cgd.ucar.edu, with a time stamp in the body of the message. If a test is already running, an email, with the subject "ESMF_Test_Builds_Pluto_not_starte is sent with "Test not started, a test is already running." in the body. The test that is running will run to completion, a new test will NOT be queued up. A new "Run_ESMF_Test_Build" email must be sent when the running test is completed.

When the test is completed an email, "ESMF_Test_Builds_Pluto" with the test results is sent to esmf-test@lists.sourceforge.net, esmf_test@cgd.ucar.edu. The test results will also appear in the Regression Test webpage under "ESMF_Test_Builds" link towards the top of the page.

4.3.6 Code Reviews

1. All significant chunks of externally contributed code are reviewed by the JST. It's usual to do the code review after check-in. The code review should be scheduled with the Core Team Manager when the code is checked in, and the code review held before the next release.
2. We also do code reviews with core team members, as desired/required by the JST.

4.3.7 Releases

The ESMF produces internal releases and public releases based on the schedule generated by the CRB. Every public release is preceded by an internal release three months prior, for the purpose of beta testing. During those three months, bugs may be fixed and documentation improved, but no new functionality may be added. Occasionally the Core Team releases an internal release that does not become a public release. This would happen, for example, when major changes are being made to ESMF and user input is needed for multiple preliminary versions of the software.

The Integrator tags new system versions with coherent changes prior to release. The tagging convention for public and internal releases is described in Section 7.20.

Prior to release all ESMF software is regression-tested on all platforms and interfaces. The Integrator is responsible for regression testing, identifying problems and notifying the appropriate developers, and collecting and sending out Release Notes and Known Bugs.

ESMF releases are announced on the esmf_jst@cgd.ucar.edu mailing list and are posted on the ESMF website. Source code is available for download from the ESMF website and from the main ESMF SourceForge site.

4.3.8 Backups

The backup strategy for each entity of the ESMF project is as follows:

- **ESMF CVS source**
  Run rsync daily on the ESMF cvs repository and roll a tarball. On Sundays roll a tarball with a date stamp and move it to the Pluto archive directory.
- **ESMF GIT source**
  Run rsync daily on the ESMF git repository and roll a tarball. On Sundays roll a tarball with a date stamp and move it to the Pluto archive directory.
- **ESMFCONTRIB CVS source**
  Run rsync daily on the ESMFCONTRIB cvs repository and roll a tarball. On Sundays roll a tarball with a date stamp and move it to the Pluto archive directory.
- **ESMF website**
  On Sundays make a tarball with a date stamp of the ESMF website and move it to the Pluto archive directory.

To conserve memory only the backup files for the current year and the prior year are retained. For years beyond the prior year, only 6 month backup files are retained i.e. for 2010 to 2012 of the ESMF cvs files are:
20100103.esmf-cvsroot.tar.gz
20100606.esmf-cvsroot.tar.gz
20110102.esmf-cvsroot.tar.gz
20110605.esmf-cvsroot.tar.gz
20120101.esmf-cvsroot.tar.gz
All of 1012 and 1013

Once a year in January, the backup files of the year before the prior year will be cleaned up. For example, In January 2014 all of backup files of 2012 and 2013 would be archived, so the 2012 backup files will be cleaned up and only 6 month backup files will be retained.

4.4 Testing and Validation

ESMF software is subject to the following tests:

1. Unit tests, which are simple per-class tests.
2. Testing Harness, parameter space spanning tests similar to the unit tests
3. System tests, which generally involve inter-component interactions.
4. Use test cases (UTCs), which are tests at realistic problem sizes (e.g., large data sets, processor counts, grids).
5. Examples that range from simple to complex.
6. Beta testing through preliminary releases.

Unit tests, system tests, and examples are distributed with the ESMF software. UTCs, because of their size, are stored and distributed separately. Tests are run nightly, following a weekly schedule, on a wide variety of platforms. Beta testing of ESMF software is done by providing an Internal Release to customers three months before public release.

The ESMF team keeps track of test coverage on a per-method basis. This information is on the **Metrics** page under the **Development** link on the navigation bar.

Testing information is stored on a **Test and Validation** web page, under the **Development** link on the ESMF web site. This web page includes:

- separate web pages for each system test and UTC;
- links to the *Developer's Guide*, SourceForge Tracker, Requirements Spreadsheet, and any other pertinent information; and
- separate web page for automated regression test information and results.

The ESMF is designed to run on several target platforms, in different configurations, and is required to interoperate with many combinations of application software. Thus our test strategy includes the following.

- Tests are executed on as many target platforms as possible.
- Tests are executed on a variety of programming paradigms (e.g pure shared memory, pure distributed memory and a mix of both).
- Tests are executed in multiple configurations (e.g. uni-processor, multi-processor).
- The result of each test is a `PASS/FAIL`.
- In some cases, for floating point comparisons, an epsilon value will be used.
- Tests are implemented for each language interface that is supported.

4.4.1 Unit Tests

Each class in the framework is associated with a suite of unit tests. Typically the unit tests are stored in one file per class, and are located near the corresponding source code in a test directory. The framework `make` system will have an option to build and run unit tests. The user has the option of building either a "sanity check" type test or an exhaustive suite. The exhaustive tests include tests of many functionalities and a variety of valid and invalid input values. The sanity check tests are a minimum set of tests to indicate whether, for example, the software has been installed correctly. It is the responsibility of the software developer to write and execute the unit tests. Unit tests are distributed with the framework software.

To achieve adequate unit testing, developers shall attempt to meet the following goals.

- Individual procedures will be evaluated with at least one unit test function. However, as many test functions as necessary will be implemented to assure that each procedure works properly.
- Developers should unit test their code to the degree possible before it is checked into the repository. It is assumed that developers will use stubs as necessary.
- Variables are tested for acceptable range and precision.
- Variables are tested for a range of valid values, including boundary values.
- Unit tests should verify that error handling works correctly.

Writing Unit Tests Unit tests usually test a single argument of a method to make it easier to identify the bug when a unit test fails. There are several steps to writing a unit test. First, each unit test must be labeled with one of the following tags:

- NEX_UTest - This tag signifies a non-exhaustive test. These tests are always run and are considered to be sanity tests they usually consist of creating and destroying a specific class.
- EX_UTest - This tag signifies an exhaustive unit test. These tests are more rigorous and are run when the ESMF_EXHAUSTIVE environmental variable is set to ON. These unit test must be between the #ifdef ESMF_EXHAUSTIVE and #endif definitions in the unit test file.

- NEX_UTest_Multi_Proc_Only - These are non-exhaustive multi-processor unit tests that will not be run when the run_unit_tests_uni or unit_test_uni targets are specified.
- EX_UTest_Multi_Proc_Only - These are exhaustive multi-proccesor unit tests that will not be run when the run_unit_tests_uni or unit_tests_uni targets are specified.

Note that when the NEX_UTest_Multi_Proc_Only or EX_UTest_Multi_Proc_Only tags are used, all the unit tests in the file must be labeled as such. You may not mix these tags with the other tags. In addition, verify that the makefile does not allow the unit tests with these tags to be run uni.

Second, a string is specified describing the test, for example:

```
write(name, *) "Grid Destroy Test"
```

Third, a string to be printed when the test fails is specified, for example:

```
write(failMsg, *) "Did not return ESMF_SUCCESS"
```

Fourth, the ESMF_Test subroutine is called to determine the test results, for example:

```
call ESMF_Test((rc.eq.ESMF_SUCCESS), name, failMsg, result, ESMF_SRCLINE)
```

The following two tests are good examples of how unit tests should be written. The first test verify that getting the attribute count from a Field returns ESMF_SUCCESS, while the second verifies the attribute count is correct. These two tests could be combined into one with a logical AND statement when calling ESMF_Test, but breaking the tests up allows you to identify the source of the bug immediately.

```
!------------------------------------------------------------------------
!EX_UTest
! Getting Attrubute count from a Field
call ESMF_FieldGetAttributeCount(f1, count, rc=rc)
write(failMsg, *) "Did not return ESMF_SUCCESS"
write(name, *) "Getting Attribute count from a Field "
call ESMF_Test((rc.eq.ESMF_SUCCESS), name, failMsg, result, ESMF_SRCLINE)

!------------------------------------------------------------------------
!EX_UTest
! Verify Attribute Count Test
write(failMsg, *) "Incorrect count"
write(name, *) "Verify Attribute count from a Field "
call ESMF_Test((count.eq.0), name, failMsg, result, ESMF_SRCLINE)

!------------------------------------------------------------------------
```

Sometimes a unit test is written expecting a subset of the processors to fail the test. To handle this case, the unit test must verify results from each processor as in the unit test below:

```
    !------------------------------------------------------------------------
    !EX_UTest
    ! Verify that the rc is correct on all pets.
    write(failMsg, *) "Did not return FAILURE  on PET 1, SUCCESS otherwise"
    write(name, *) "Verify rc of a Gridded Component Test"
    if (localPet==1) then
      call ESMF_Test((rc.eq.ESMF_FAILURE), name, failMsg, result, ESMF_SRCLINE)
    else
      call ESMF_Test((rc.eq.ESMF_SUCCESS), name, failMsg, result, ESMF_SRCLINE)
    endif

    !------------------------------------------------------------------------
```

Some tests may require that a loop be written to verify multiple results. The following is an example of how a single tag, NEX_UTest, is used instead of a tag for each loop iteration.

```
!-------------------------------------------------------------------------
 !NEX_UTest
 write(name, *) "Verifying data in Array via Fortran array pointer access"
 write(failMsg, *) "Incorrect data detected"
 looptest = .true.
 do i = -12, -6
   j = i + 12 + lbound(fptr, 1)
   print *, fptr(j), fdata(i)
   if (fptr(j) /= fdata(i)) looptest = .false.
 enddo
 call ESMF_Test(looptest, name, failMsg, result, ESMF_SRCLINE)
 !-------------------------------------------------------------------------
```

Analyzing unit test results When unit test are run, a Perl script prints out the test results as shown in Section "Running ESMF Unit Tests" in the ESMF User's Guide. To print out the test results, the Perl script must determine the number of unit tests in each test file and the number of processors executing the unit test. It determines the number of tests by counting the EX_UTest, NEX_UTest, EX_UTest_Multi_Proc_Only, or NEX_UTest_Multi_Proc_Only tags in the test source file whichever is appropriate for the test being run. To determine the number of processors, it counts the number of "NUMBER_OF_PROCESSORS" strings in the unit test output Log file. The script then counts the number of PASS and FAIL strings in the test Log file. The Perl script first divides the number of PASS strings by the number of processors. If the quotient is not a whole number then the script concludes that the test crashed. If the quotient is a whole number, the script then divides the number of FAIL strings by the number of processors. The sum of the two quotients must equal the total number of tests, if not the test is marked as crashed.

Disabling unit tests Sometimes in the software development process it becomes necessary to disable one or more unit tests. To disable a unit test, two lines need to be modified. First, the line calling "ESMF_Test" must be commented out. Second, the NEX_UTest, EX_UTest, NEX_UTest_Multi_Proc_Only and EX_UTest_Multi_Proc_Only tags must be modified so that they are not found by the Perl script that analyzes the test results. The recommended way to modify these tags is to replace the first underscore with "_disable_", thus NEX_UTest becomes NEX_disable_UTest.

Benchmarking Unit Tests Benchmark testing is included in the ESMF regression tests to detect any unexpected change in the performance of the software. This capability is available to developers. Developers can run the unit tests and save their execution times to be used as a benchmark for future unit test runs.

The following section now appears in the output of "gmake info".

```
------------------------------------------------------------
 * ESMF Benchmark directory and parameters *
ESMF_BENCHMARK_PREFIX:    ./DEFAULTBENCHMARKDIR
ESMF_BENCHMARK_TOLERANCE: 3%
ESMF_BENCHMARK_THRESHOLD_MSEC: 500

------------------------------------------------------------
```

The steps for using the benchmarking test tool are as follows:

- After building the unit tests, execute "gmake run_unit_tests" and verify that all tests pass. It not recommended that failing tests be benchmarked.
- Set "BENCHMARKINSTALL = YES" and execute "gmake run_unit_tests_benchmark". This will cause the unit tests stdout files to be copied to the "DEFAULTBENCHMARKDIR" directory. The elapsed times of these unit tests are the now the benchmark. The default of DEFAULTBENCHMARKDIR is $ESMF_DIR/DEFAULTBENCHMARKDIR. It is advised that the benchmarking directory be outside the ESMF structure, to allow the developer to benchmark different versions of the software. The benchmark directory can be changed by setting ESMF_BENCHMARK_PREFIX.
- Run the unit test a second time.
- To compare the elapsed times of this unit test run to the benchmared run, set "BENCHMARKINSTALL = NO", and execute "gmake run_unit_tests_benchmark".

According to the default settings above, the benchmarking test will only analyze unit tests that run 500 msecs (ESMF_BENCHMARK_THRESHOLD_MSEC) or longer. If a unit test runs 3 percent (ESMF_BENCHMARK_TOLERANCE) or more beyond the benchmarked unit test, it will be flagged as failing the benchmark test. The developer may change these parameters as desired. The following is an example of the output of running "gmake run_unit_tests_benchmark":

```
The following unit tests with a threshold of 500 msecs. passed the 3%
 tolerance benchmark test:

PASS: src/Infrastructure/DELayout/tests/ESMF_DELayoutWorkQueueUTest.F90
PASS: src/Infrastructure/Field/tests/ESMF_FieldCreateGetUTest.F90
PASS: src/Infrastructure/Field/tests/ESMF_FieldRegridCsrvUTest.F90
PASS: src/Infrastructure/Field/tests/ESMF_FieldRegridXGUTest.F90
PASS: src/Infrastructure/Field/tests/ESMF_FieldStressUTest.F90
PASS: src/Infrastructure/TimeMgr/tests/ESMF_CalRangeUTest.F90
PASS: src/Infrastructure/VM/tests/ESMF_VMBarrierUTest.F90
PASS: src/Infrastructure/VM/tests/ESMF_VMUTest.F90
PASS: src/Infrastructure/XGrid/tests/ESMF_XGridMaskingUTest.F90
```

```
PASS: src/Infrastructure/XGrid/tests/ESMF_XGridUTest.F90
PASS: src/Superstructure/Component/tests/ESMF_CompTunnelUTest.F90


The following unit tests with a threshold of 500 msecs. failed the 3%
 tolerance benchmark test:

FAIL: src/Infrastructure/Field/tests/ESMF_FieldRegridUTest.F90
      Test elapsed time: 4331.446 msec.
      Benchmark elapsed time: 2958.47675 msec.
      Increase: 46.41%

FAIL: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleRegridUTest.F90
      Test elapsed time: 2051.05675 msec.
      Benchmark elapsed time: 1920.42125 msec.
      Increase: 6.8%

FAIL: src/Infrastructure/LogErr/tests/ESMF_LogErrUTest.F90
      Test elapsed time: 2986.40425 msec.
      Benchmark elapsed time: 2583.36775 msec.
      Increase: 15.6%



Found 167 exhaustive multi-processor unit tests files, of those with a
 threshold of 500 msecs. 11 passed the 3% tolerance benchmark test, and 3 failed.

Benchmark install date: Thu Jun  4 13:26:55 MDT 2015
```

Note that only the unit tests that have an elapsed time of 500 msecs. or greater are listed. In addition, the date when the benchmark install was completed is displayed.

When a unit test run it benchmarked it is written to a directory such as "BENCHMARKDIR/test/testg/Darwin.gfortran.64.mpich2.default/". Therefore you can only compare unit tests elapsed between the identical configurations.

To implement the benchmarking tool, the unit tests were modified to record the elapsed time of each PET. The stdout file of each unit test has the following lines i.e.

```
ESMF_GridItemUTest.stdout: PET 0 Test Elapsed Time 5.7840000000000007 msec.
ESMF_GridItemUTest.stdout: PET 1 Test Elapsed Time 5.7259999999999982 msec.
ESMF_GridItemUTest.stdout: PET 2 Test Elapsed Time 6.6200000000000010 msec.
ESMF_GridItemUTest.stdout: PET 3 Test Elapsed Time 5.7190000000000021 msec.
```

The benchmarking tool uses the average of the four elapsed times to determine the test results since the elapsed times of each PET can vary.

4.4.2 Examples

The examples are written to help users understand a specific use of an ESMF capability. The examples appear as text in the ESMF Reference Manual, therefore care must be taken to insure that correct portions of the examples appear in the document. Latex tags have been created to designate which portions of the examples are visible in the document.

BOE and EOE are used between text describing the example. BOC and EOC are used between actual working code that appears in the Reference Manual. Below is an example of how the tags are used:

```
!---------------------------------- Example ------------------------------
!>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%
!BOE
!\subsubsection{Get Grid and Array and other information from a Field}
!\label{sec:field:usage:field_get_default}
!
!  A user can get the internal {\tt ESMF\_Grid} and {\tt ESMF\_Array}
!  from a {\tt ESMF\_Field}.  Note that the user should not issue any destroy command
!  on the retrieved grid or array object since they are referenced
!  from within the {\tt ESMF\_Field}. The retrieved objects should be used
!  in a read-only fashion to query additional information not directly
!  available through the {\tt ESMF\_FieldGet()} interface.
!
!EOE

!BOC
    call ESMF_FieldGet(field, grid=grid, array=array, &
        typekind=typekind, dimCount=dimCount, staggerloc=staggerloc, &
        gridToFieldMap=gridToFieldMap, &
        ungriddedLBound=ungriddedLBound, ungriddedUBound=ungriddedUBound, &
        totalLWidth=totalLWidth, totalUWidth=totalUWidth, &
        name=name, &
        rc=rc)
!EOC
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
    print *, "Field Get Grid and Array example returned"

    call ESMF_FieldDestroy(field, rc=rc)
    if(rc .ne. ESMF_SUCCESS) finalrc = ESMF_FAILURE
!>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%>%
```

Note that any code or text that is not contained within the tag pairs does not appear in the Reference Manual.

Most examples can be run on multiple processors or a single processor. Those examples should have the tag, "ESMF_EXAMPLE" as a comment in the body of the example file. If the example can only run on multiple processors then use the tag, "ESMF_MULTI_PROC_EXAMPLE".

Disabling examples When an example is removed from the makefile, the "ESMF_EXAMPLE" or "ESMF_MULTI_PROC_EXAMPLE" tags must be modified so that the example is not flagged as failed. The recommended way to modify these tags is to replace the first underscore with "_disable_", thus "ESMF_EXAMPLE" becomes "ESMF_disable_EXAMPLE".

4.4.3 System Tests

System tests are written to test functionality that spans several classes. The following areas should be addressed in system testing.

- Design omissions (e.g. incomplete or incorrect behaviors).

- Associations between objects (e.g. fields, grids, bundles).
- Control and infrastructure. (e.g. couplers, time management, error handling).
- Feature interactions or side effects when multiple features are used simultaneously.

The system tester should issue a test log after each software release is tested, which is recorded on the **Test and Validation** web page. The test log shall include: a test ID number, a software release ID number, testing environment descriptions, a list of test cases executed, results, and any unexpected events. Bugs should be documented in the *SourceForge Bug Tracker* and any bug fixes shall be validated.

Writing System Tests System tests should contain the following sections:

- Create - Create Components, Couplers, Clock, Grids, States etc.
- Register - Register Components and the initialize, run and finalize subroutines.
- Initialize - Initialize as needed.
- Run - Run the test.
- Finalize - Verify results.
- Destroy - Destory all classes.

Most system tests can be run on multiple processors or a single processor. Those system tests should have the tag, "ESMF_SYSTEM_TEST" as a comment in the body of the system test. If the system test can only run on multiple processors then use the tag, "ESMF_MULTI_PROC_SYSTEM_TEST".

At the end of the system it is recommended that the ESMF_TestGlobal subroutine be used to gather test results from all processors and print out a single PASS/FAIL message instead of individual PASS/FAIL messages from all the processors. After the test is written it must be documented on the ESMF Test & Validation web page:

`http://www.earthsystemmodeling.org/developers/test/system/`

Disabling system tests When a system test is removed from the makefile, the "ESMF_SYSTEM_TEST" or "ESMF_MULTI_PROC_SYSTEM_TEST" tags must be modified so that the system test is not counted as failed. The recommended way to modify these tags is to replace the first underscore with "_disable_", thus ESMF_SYSTEM_TEST becomes ESMF_disable_SYSTEM_TEST.

4.4.4 Test Harness

The Test Harness is a highly configurable test control system for conducting thorough testing of the Regridding and Redistribution processes. The Test Harness consists of a single shared executable and a collection of customizable resource files that define an ensemble of test configurations tailored to each ESMF class. The Test Harness is integrated into the Unit test framework, enabling the Test Harness to be built and run as part of the Unit tests. The test results are reported to a single standard-out file which is located with the unit test results.

See section 6 for a complete discussion of the test harness.

Analyzing Test Harness results When the Test Harness completes a run, the results from the ensemble of tests are reported in two ways. The first is analogous to the unit test reporting, since the test harness is run as part of the unit tests, a summary of the results are recorded just as with the unit tests. In addition to the standard unit test reporting, the test harness is also able to produce a human readable report. The report consists of a concise summary of the test configuration along with the test results. The test configuration is described in terms of the Field Taxonomy syntax and user provided strings. The intent is not to provide a exhaustive description of the test, but rather to provide a useful description of the failed tests.

Consider another example similar to the previous one, where two descriptor strings describing an ensemble regridding tests. The first uses the patch method and the second uses bilinear interpolation.

```
[ B1 G1; B2 G2 ] =P=> [ B1 G1; B2 G2 ]
[ B1 G1; B2 G2 ] =B=> [ B1 G1; B2 G2 ]
```

Suppose the associated specifier files indicate that the source grid is rectilinear and is 100 X 50 in size. The destination grid is also rectilinear and is 80 X 20 in size. Both grids are block distributed in two ways, 1 X NPETS and NPETS X 1. And suppose that the first dimension of both the source and destination grids are periodic. If the test succeeds for the bilinear regridding, but fails for one of the patch regridding configurations, the reported results could look something like

```
SUCCESS: [B1 G1; B2 G2 ] =B=> [B1 G1; B2 G2 ]
FAILURE: [B1{1} G1{100}+P; B2{npets} G2{50} ] =P=> [B1{1} G1{80}+P; B2{npets} G2{20} ]
     failure at line 101 of test.F90
SUCCESS: [ B1{npets} G1{100} +P; B2{1} G2{50} ] =P=> [ B1{npets} G1{80}+P; B2{1} G2{20} ]
```

The report indicates that all the test configurations for the bilinear regridding are successful. This is indicated by the key word SUCCESS which is followed by the successful problem descriptor string. Since all of the tests in the first case pass, there is no need to include any of the specifier information. For the second ensemble of tests, one configuration passed, while the other failed. In this case, since there is a mixture of successes and failures, the report includes specifier information for all configurations to help indicate the source of the test failure. The supplemental information, while not a complete problem description since it lacks items such as the physical coordinates of the grid and the nature of the test field, includes information crucial to isolating the failed test.

4.4.5 Use Test Cases (UTCs)

Use Test Cases are problems of realistic size created to test the ESMF software. They were initiated when the ESMF team and its users saw that often ESMF capabilities could pass simple system tests but would fail out in the field, for real customer problems. UTCs have realistic processor counts, data set sizes, and grid and data array sizes. UTCs are listed on the **Test & Validation** page of the ESMF website. They are not distributed with the ESMF software; instead they are stored in a separate module in the main repository called `use_test_cases`.

4.4.6 MAPL Testing

MAPL is a software layer that establishes usage standards and software tools for building ESMF compliant components. The MAPL User's Guide can be found *here.* MAPL is included in the ESMF distribution, below are the installation and testing instructions.

Before MAPL can be installed and tested, ESMF has to be built with NetCDF and installed. Refer to the ESMF User's Guide for information on building ESMF with NetCDF. After ESMF is installed successfully, do the following:

```
Using "setenv" or "export" depending on the shell being used:
set ESMFMKFILE = $ESMF_DIR/lib/lib$ESMF_BOPT/$ESMF_OS.$ESMF_COMPILER.
                         $ESMF_ABI.$ESMF_COMM.$ESMF_SITE/esmf.mk.
```

```
To install MAPL, enter "gmake install_mapl".

To build the MAPL tests, enter "gmake build_mapl_tests".

To run the tests, enter "gmake run_mapl_tests" or "run_mapl_test_uni".
```

The MAPL tests follow the same structure and testing options as the ESMF unit tests, see section **Unit Tests** of this document for details.

4.4.7 Beta Testing

ESMF software is released in a beta form, as an Internal Release, three months before it is publicly released. This gives users a chance to test the software and report back any problems to support.

4.4.8 Automated Regression Tests

The purpose of regression testing is to reveal faults caused by new or modified software (e.g. side effects, incompatibility between releases, and bad bug fixes). Regression tests regularly exercise all interfaces of the code on all target platforms. The regression test results for the last two weeks can be found *here.* This web page provides a complete color-coded current view of the state of the trunk ESMF software, sorting options by platform or compiler are provided. A similiar test results web page for the branch is also available. Clicking on any of the cells will display the specific test report for that day. Hovering over the test name i.e., Blues gfortran, will reveal notes particular to that platform/compiler. Clicking on the test name, will take you to the home page of the platform.

The platforms that run the regression tests, email the test results to a server that updates the test results web page. A script checks for test reports every 15 minutes, and updates the web page. The time of the last update appears on the web page.

4.4.9 Investigating Test Failures

The regression test results web *page* provides a color-coded view of the state of the software. When a developer finds that a test fails on a particular platform with a particular compiler, sometimes the bug is readily identified and fixed. However other times the developer may want to know if the test fails on other platforms and if the failure is related to a compiler, mpi configuration or optimized/debug execution. The developer would need to click to all the cells of different platforms searching for the test results for that particular test.

A tool was created to allow the developers to query the test results for a specific test for a specific date, as long as it is within two weeks of the current date. The developer may send a query test results message to the following email address: esmftest@cgd.ucar.edu The subject of the email must be exactly "Test_Results_Query". The body of the email message must be "Test:" followed by the test name and "Date" followed by the desired date. The format must be a three letter month and a number. If the date is 2 digits, greater than 9, then insert one space between the month and date e.g. Apr 25. If the day is a single digit insert two spaces, between the month and day e.g. Apr 4.

```
Test:ESMF_FieldBundleSMMUTest.F90
Date:Feb  8
   or
Date Feb 28
```

This mail box is checked every quarter hour on the quarter hour, the results are emailed to:esmf_test@cgd.ucar.edu. The subject of the results email for this example would be:

```
ESMF_FieldBundleSMMUTest.F90 test results for Feb  8
```

The body of the email would be as follows:

```
ESMF_Blues_PGI:PASS: mvapich2/g: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUT
ESMF_Blues_PGI:PASS: mvapich2/O: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUT
ESMF_Blues_PGI:CRASHED: mpich3/g: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMU
ESMF_Blues_PGI:PASS: mpich3/O: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUTes
ESMF_Blues_PGI:PASS: openmpi/g: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUTe
ESMF_Blues_PGI:PASS: openmpi/O: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUTe
ESMF_Discover_g95:PASS: mvapich2/g: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSM
ESMF_Discover_g95:PASS: mvapich2/O: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSM
ESMF_Haumea_g95:PASS: mpich2/g: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUTe
ESMF_Haumea_g95:PASS: mpich2/O: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUTe
ESMF_Haumea_g95:PASS: mvapich2/g: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMU
ESMF_Haumea_g95:PASS: mvapich2/O: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMU
ESMF_Pluto_g95:FAIL: mpich2/g: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUTes
ESMF_Pluto_g95:FAIL: mpich2/O: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUTes
ESMF_Pluto_g95:FAIL: mvapich2/g: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUT
ESMF_Pluto_g95:FAIL: mvapich2/O: src/Infrastructure/FieldBundle/tests/ESMF_FieldBundleSMMUT
```

Note that if the date of the query is the current day, the developer should query periodically during the day since the test results are being updated as platforms report their test results. If a test crashes it can be because another test hung and the test in question did not run.

Another instance where this tool is useful is when a developer adds a new test, after the nightly tests run, the developer can run a query to quickly see the test results.

4.4.10 Building the Documentation

As software development progresses, the documentation is updated, built and posted at *https://www.earthsystemcog.org/projects/esmf/dev*

The documents are built daily in the early morning, the results of the builds are posted at *http://www.earthsystemmodeling.org/develop*

These documents can be updated by the developers, by checking out the documents from the repository and submitting the edited files. To have the new version of the documents posted on the web, the developer must sent a request to the following email address: esmftest@cgd.ucar.edu. The subject of the email indicates which document to build and post. The following is the list of subjects that have been implemented:

- Build_Dev_Guide_Doc
- Build_NUOPC_Doc
- Build_Ref_Doc
- Build_ESMPy_Doc
- Build_CICE_NUOPC_CAP_Doc
- Build_HYCOM_NUOPC_CAP_Doc
- Build_LIS_NUOPC_CAP_Doc
- Build_MOM_NUOPC_CAP_Doc
- Build_WRFHYRO_NUOPC_CAP_Doc

A script checks for document build requests every quarter hour on the quarter hour. A document build is started and on successful completion the document is updated on the web and document build results is updated. An email will be sent to esmf_test@cgd.ucar.edu and esmf-test@lists.sourceforge.net when the build is done.

4.4.11 Testing for Releases

We provide two types of tar files, the ESMF source and the shared libraries of the supported platforms. Consequently, there are two test procedures followed before placing the tar files on the ESMF download website.

The **Source Code Test Procedure** is followed on all the supported platforms for the particular release.

1. Verify that the source code builds in both `BOPT=g` and `BOPT=O`.
2. Verify that the `ESMF_COUPLED_FLOW` demonstration executes successfully.
3. Verify that the unit tests run successfully, and that there are no `NON-EXHAUSTIVE` unit tests failures.
4. Verify that all system tests run successfully.

The **Shared Libraries Test Procedure** is also followed on all supported platforms for a release.

1. Change to the `CoupledFlowEx` directory and execute `gmake`. Verify that the demo runs successfully.
2. Change to the `CoupledFlowSrc` directory and execute `gmake` then `gmake run`. Verify that the demo runs successfully.
3. Change to the `examples` directory and execute `gmake` and `gmake run`. Verify that the example runs successfully.

4.5 User Support

4.5.1 Roles

The **Advocate** is the staff person assigned to a particular code e.g. GEOS-5. See section 2.1.1 for a full definition and list of responsibilities. The **Handler** is the staff person assigned to solve a support ticket. The Advocate and the Handler may be the same person or they may be different. See section 2.1.1 for complete definition and list of responsibilities.

4.5.2 Support Categories

New is a request that has not been replied to. **Closed** is a request that has been fixed to the user's satisfaction. **Pending** is a request that has been fixed to the Handler's satisfaction but has not yet been approved by the user.

4.5.3 Summary Work Flow

1. Message received.
2. The Integrator or in his absence the Support Lead, generates a SourceForge Bug, Feature, or Support Request ticket.
3. If the request contains more than one topic, then Integrator will open multiple tickets, one per topic. This can been done initially if obvious, or later if more research indicates it is necessary.
   - The top line of the entry should be WHO: <Requester Name>.
   - Indicate the institution and model if known.
   - Keep title of initial email and the title of the SF ticket the same or close enough to be able to determine they are one and the same.
   - Assign the ticket to the staff person best able to solve the ticket's issue.
4. Initial contact is made by:

- The Handler assigned by the Integrator in the ticket.
- The Support Lead if the Handler will be unavailable for more than a week.

5. The Handler works to solve the tickets issues. He or she will communicate periodically with the ticket's originator and will keep the rest of the Core team informed on the tickets progress at the monthly ticket review meetings. Once the issue has been solved, the ticket will be marked pending by the Handler.
6. At this point, the Handler contacts the originator to gauge their satisfaction with the solution. If the originator is satisfied, the ticket may be closed, and the mail folder on the IMAP server moved from Open to Closed by the Support Lead. If the customer does not respond, an attempt at contact will be made once a month for two months. If after this period, the originator still does not reply, a pending ticket may be closed with final notification to the originator.

4.5.4 General Guidelines for Handling Tickets

- Include title and ticket number on all correspondence.
- Make initial contact within 48 hours even if just to say message received.
- The email address for ticket originators can be found in either freeCRM or the mail archive. Do not hesitate to contact the Support Lead if a required email address can not be found.
- Copy esmf_support@ucar.edu on all replies.
- Bugs that are fixed should be marked Closed, and Fixed. They should never be deleted.
- Bugs that are duplicates should be marked Deleted, and Duplicate.
- If the main issue in a Bug, Feature Request, or Support Request has not been implemented it should stay Open.
- Users are always notified via email when their ticket is being closed even if they have been unresponsive.
- If the solution to a ticket involves a test code, this should be incorporated into the code body as standard test. It should not be sent to the user as an unofficial code fragment.
- If the solution to a ticket involved changes to the code, the user should be given a stable beta snapshot. The user should not be directed to the HEAD, which is inherently unstable.
- If a ticket involves an older version of the code and a computing environment that the current distribution runs on, the ticket should be considered for closure when there is no means of testing or fixing the older code.
- The Handler is responsible for changing the status of tickets assigned to them.

4.5.5 esmf_support@ucar.edu Mail Archives

The Support Lead manages the archive of esmf_support@ucar.edu email traffic and is responsible for the creation of ticket folders, component folders, and the proper placement of mail messages. The archive is located on the main CISL IMAP server and can be accessed by any Core member. Contact the Support Lead if you wish your local mail client enabled to view the archive. The IMAP archive will have the following appearance:

- Component Name
  - Open
    - Numbered Ticket Folder
    - Numbered Ticket Folder
    - Numbered Ticket Folder

  - Closed
    * Numbered Ticket Folder
    * Numbered Ticket Folder
    * Numbered Ticket Folder

- Component Name

The following rules apply to the above:

- Email messages will be filed by component and number.
- A folder labeled with the request number will be created.
- This folder will then be placed in the components Open folder until closed.
- The Support Lead will copy each related email message to its numbered folder.
- When a ticket has been closed, the Support Lead will move the numbered folder from the components Open folder to its Closed folder.
- There will be only one New folder to which highly active tickets may be placed for easier filing at the discretion of the Support Lead.

4.5.6 INFO:Code (subject) mail messages

Advocates need to share the information they have received from their codes with the rest of the Core team. This will be done by sending an email to esmf_support@ucar.edu with a subject line labeled INFO: Code e.g. INFO: CCSM, INFO: GEOS-5. These messages will be filed on the IMAP server (see above section) under the code referenced. All information about a code that is general and not related to a specific support request will be archived in this manner.

4.5.7 freeCRM

A client relationship management tool (freeCRM http://www.freecrm.com) is being used to archive codes, their affiliated contacts, degree of componentization, issues, and applicable funding information if known. The following is a list of roles and responsibilities associated with this software:

- Advocates are responsible for the accuracy and completeness of all information associated with codes to which they are assigned. This information includes a pull down menu that specifies the state of the code's ESMF'ization. This piece of information is critical and needs to be updated whenever an Advocate updates his or her codes. Other information includes type of code, parent agency etc. This information will be reviewed on a semi-annual basis.
- The Integrator is responsible for creating a back up of all freeCRM data on a monthly basis.
- The Core Team Manager is responsible for the accuracy and completeness of all funding related information.
- The Support Lead is responsible for creating code 'companies' and informing the Integrator of any additions so that the back up scripts can be modified. He or she is additionally responsible for conducting semi-annual quality control checks of all information in the system.
- All team members are responsible for updating and adding to the list of contacts.

4.5.8 Annual Code Contact

Once a year all codes in the freeCRM data base will be contacted in order to gauge their development progress, and to update our component metrics. This process will contain the following steps:

- Advocates will login into freeCRM and get a list of all their codes. This list will be emailed to esmf_core@cgd.ucar.edu.
- Advocates will review their list and determine which codes on the list need to be contacted. Contact is not needed if sufficient knowledge is already known about a code.
- Advocates will review all the information contained in freeCRM concerning their assigned codes AND review all the esmf_support traffic for the last year.
- Advocates will draft the contact email and send it to esmf_core@cgd.ucar.edu to be reviewed by the Core Team Manager. Once corrected, the Advocates may send their email. Since this is a group level effort, the email message may be signed "The ESMF Team" if desired.
- The Support Lead will track the draft and completed emails as well as the responses and will provide a report to the Core Team Manager at the end of the process.
- As responses come in, the Advocates are responsible for updating the information in freeCRM.
- The Support Lead will tally the results and update the components page on the ESMF Web site and will also update the components metric chart.

4.5.9 Dealing with Applications that use ESMF

More and more applications are being distributed with embedded ESMF interfaces. It may difficult to determine if a reported problem with one of these applications is related to an incorrect ESMF implementation, a true ESMF bug, or an issue within the parent model. The following are several definitions:

- End User: A person who downloads or otherwise receives an application that contains ESMF code. While they may be trying to modify this application, they were not the person or persons who originally inserted ESMF into the application. Most likely, they will be entirely unfamiliar with ESMF.
- Application Developer: The person or persons who took a model, inserted ESMF code, and made the resulting application available to others.

The following are some guidelines for dealing with such Applications that use ESMF:

- For support requests related to applications that include ESMF, our primary contact for resolving the request should be the developers of the distributed application and not the End User. As such, every effort should be made to identify and contact the developers of the distributed application in order to make them aware of the reported issue and to get them actively engaged in resolution of the problem. Additionally, they should be cc'ed on all correspondence with the End User.
- During the resolution of the issue, it will be necessary to cc all email traffic to the End User. In dealings with the End User emphasize that the ESMF group is committed to any user of ESMF regardless of source. That commitment is predicated, however, on participation of the application developers.
- The Handler should establish which version of ESMF the application is using.
- The Handler should try and determine whether the ESMF code in question was modified in any way by the Application Developers.
- The Handler should try and determine whether the code in question has ESMF interface names but is not ESMF code. The time manager in WRF falls into this category. It has ESMF interfaces but was not developed by us.

- It will be solely the Core Team's discretion whether or not to support older versions of ESMF, ESMF code that has been modified by others, or code that uses ESMF interface names but was developed entirely separate from ESMF.
- In no way should the Handler try running the End User's code.
- In the event that the developers of the distributed application are unknown, unreachable, or uncooperative, the End User must be politely informed that the group can not troubleshoot code belonging to another group. This will have to been handled with a degree of sensitivity because it is likely that the end user has already tried to contact the application developers without success.

5 Apps

The ESMF source distribution contains a few executables that are built on top of the ESMF library during installation. These executables are referred to as *apps*. Apps are command line tools intended to be used by the end user of ESMF. ESMF follows the GNU conventions for command line tools. There are currently three mandatory command line options that every distributed app must implement:

```
--help      Print a help message, documenting all of the available command line arguments a
--version   Print the version of ESMF in long format and the copyright message.
-V          Print just the version string of ESMF.
```

6 Test Harness

The Test Harness is a flexible test control system intended to provide a thorough parameter space exploration of remapping and redistribution of distributed arrays and fields. The parameter space is defined through configuration files which are interpreted at run time to perform the desired tests.

The Test Harness is integrated into the Unit test framework, enabling the Test Harness to be built and run as part of the Unit tests. The test results are reported to a single standard-out file which is located with the unit test results.

The motivation for employing such a hierarchy configuration files is to allow a high degree of customization of the test configurations by combining individual specification files. Complex combinations of test cases are easily specified in high level terms. Each class will have its own collection of specification files tailored to the needs of that class.

6.1 Specifying Test Harness Tests

The test harness code consists of a single executable that uses a customizable configuration files which are located within the `<classdir>/tests/harness_config` directory of each supported class; currently only ESMF_ARRAY and ESMF_FIELD.

There are three ways to invoke the test harness, as an integral part of running unit tests; as a stand-alone test invoked through gmake; and as a standalone test invoked through the command line.

Running the harness along with the unit tests provides frequent regression testing of the redistribution and regridding features.

Running the test harness in stand alone mode using gmake is useful for isolating faults in failed test cases.

Running the test harness from the command line provides the most control over Test Harness execution. Understanding the underlying program allows the developer full access to test harness features. This is useful in developing makefiles, scripts, and configuration files.

6.2 Running Test Harness Tests as Part of Unit Tests

When running as an integral part of the unit tests, the makefile contained in the `<classdir>/tests` directory of each supported class is executed through the run_unit_tests target. As part of this target, the makefile selects the desired test harness target using a class specific environment variable (e.g. ESMF_TESTHARNESS_FIELD). A default case is provided in case the environment variable variable is not set.

The environment variables currently defined are:

- ESMF_TESTHARNESS_ARRAY - set target for Array class tests
- ESMF_TESTHARNESS_FIELD - set target for Field class tests

The targets currently supported for the ESMF_TESTHARNESS_ARRAY variable are:

- RUN_ESMF_TestHarnessArray_default - Used to verify functionality of the test harness
- RUN_ESMF_TestHarnessArray_1 - Basic test of 2D and 3D redistribution
- RUN_ESMF_TestHarnessArray_2 - Quick test of 2D and 3D redistribution
- RUN_ESMF_TestHarnessArrayUNI_default - Used to verify functionality of the test harness in uni-PET mode
- RUN_ESMF_TestHarnessArrayUNI_1 - Basic test of 2D and 3D redistribution in uni-PET mode
- RUN_ESMF_TestHarnessArrayUNI_2 - Quick test of 2D and 3D redistribution in uni-PET mode

The targets currently supported for the ESMF_TESTHARNESS_FIELD variable are:

- RUN_ESMF_TestHarnessField_default - Used to verify functionality of the test harness
- RUN_ESMF_TestHarnessField_1 - Basic test of 2D and 3D regrid
- RUN_ESMF_TestHarnessFieldUNI_default - Used to verify functionality of the test harness in uni-PET mode
- RUN_ESMF_TestHarnessFieldUNI_1 - Basic test of 2D and 3D regrid in uni-PET mode

Each target selects the desired sequence of test cases for that target. For example, a series of test cases could be developed to run on different days providing partial test coverage on a particular day, but complete coverage over a week. In this case, each day would have a separate target and the environment variable would be set for that day's test case.

An example of a test harness appears below.

```
RUN_ESMF_TestHarnessField_1:
\$(MAKE) TESTHARNESSCASE=field_1 NP=4 run_test_harness
```

In this example, the test harness target is RUN_ESMF_TestHarnessField_1. This target will execute a single test harness test case. Additional lines can be added if additional steps are desired. The next section will describe the details of invoking a test case.

6.3 Invoking a Single Test Harness Test Case Using gmake

A single test harness test case can be invoked from the `<classdir>/tests` directory through the make command with the local parameters `TESTHARNESSCASE` and `NP` with the `run_test_harness` target.

For example,

```
gmake TESTHARNESSCASE=field_1 NP=4 run_test_harness
```

will run the test harness test case, `field_1` on `4` processors.

Each test case is defeined by a series of configuration files in the `<classdir>/tests/harness_config` directory. All of the configuration files for a particular test case will be prefixed with `<casename>_` where <classname> is a unique name for the test. The top level configuration file will have a suffix of `_test.rc`. Thus, the top level configuration file for the `field_1` test case will be `field_1_test.rc`.

The next section describes invoking the Test Harness from the Command Line.

6.4 Invoking a Single Test Harness Test Case from the Command Line

The Test Harness is built by invoking gmake from the $ESMF_DIR/src/test_harness/src directory or by building all the unit tests. This creates the exececutable file ESMF_TestHarnessUTest in the currently selected test directory.

To run the executable, enter:

```
\<run path\>ESMF\_TestHarnessUTest \<cmd args\>
where,
\<run path\> is the path to the executable,
and \<cmd args\> are the optional command arguments

The cmd args are as follows:
-path \<config path\>
-case \<case filename\>
-xml \<xml filename\>
-norun
```

The path argument sets the path to the configuration files. All of the configuration files for a testcase must reside in the same directory. If this argument is not present, the current working directory will be used as the path.

The case argument is the name of the top level configuration file. This is described in a later section of this document. If this argument is not present, test_harness.rc will be used.

The xml argument instructs the test harness to generate an XML test case summary file in the current working directory.

The norun argument instructs the test harness not to run the test cases. The configuration files will be parsed and if selected, an XML file will be generated. The XML file can be post-processed to generate human readable test configuration summary reports.

If running under mpi, you would need to prefix this command with the proper mpirun command.

The next section describes the format of the top level resource file.

6.5 Top Level Configuration File

As mentioned above, the top level configuration file will be located in the `<classdir>/tests/harness_config` directory and named `<casename>_test.rc`.

The top level configuration file specifies the test class, the format for reporting the test results, and the location and file names containing the *problem descriptor files*.

Originally, the test harness had a single configuration file for each class and supported only two test cases, a non-exhaustive case and an exhaustive case. This turned out to be too restrictive and test cases are now selected through environment variables. Unfortunately, some of the old constructs remain until the obsolete feature is removed.

Currently, the test harness only uses the non-exhaustive test case for each configuration file. So, referencing the example below, only the nonexhaustive tag is actually used by the test harness.

Also note, that comments are preceded by the # sign, that single parameter values follow a single colon : punctuation mark, while tables of multiple parameter values follow, and are terminated by, a double set of colon :: punctuation marks. This file is read by the `ESMF_Config` class methods, therefore must adhere to their specific syntax requirements. The entries can be in any order, but the name tags must be exact - including CAPITALIZATION. While it is not strictly necessary, the file names are enclosed in quotation marks, either single or double, to guarantee they are read correctly.

```
# Field test Harness Config file

# test class
test_class: FIELD
```

```
# report a summary of the test configurations before actually conducting tests
setup_report: TRUE

# test result report - options are:
# test_report: FULL  - full report presenting both success and failure configs
# test_report: FAILURE - report only failure configurations
# test_report: SUCCESS - report only successful configurations
# test_report: NONE - no report
test_report: FAILURE

# descriptor file for the nonexhaustive case
nonexhaustive::
   'nonexhaustive_descriptor.rc'
::   # end of list

# descriptor files for the exhaustive case (obsolete, but keep in for awhile)
exhaustive::
   'exhaustive_descriptor.rc'
:: # end of list
```

The argument for the tag `test_class:` specifies the ESMF class to be tested. Here it is Field.The tag `setup_report:` specifies if a setup report is sent to the test report. The tag `test_report:` specifies the style of report to be constructed. This report is appended to the standard-out file which is located with the Unit test results. The tag `nonexhaustive::` delimits a table which contains the file names of problem descriptor files pertaining to the current test configuration.

The tag exhaustive is obsolete and will be removed when time permits. It has been replaced with the Test Harness environment variables which can select the desired test case from the available portfolio.

6.5.1 Problem descriptor file

The problem descriptor files contain *descriptor strings* that describe the family of problems with the same memory topology, distribution, and grid association. The files also contain the names of *specifier files* which complete the descriptions contained within the descriptor strings. This structure allows a high level of customization of the test suite.

The problem descriptor file contains only one table, again conforming to the `ESMF_Config` class standard. The contents of the table must be delimited by the tag `problem_descriptor_string::`. The first element on the line, enclosed by quotes, is the problem descriptor string itself. Since the descriptor strings contain gaps and special characters, it is necessary to enclose the strings in quotation marks to guarantee proper parsing. The problem descriptor table may contain any number of descriptor strings, up to the limit imposed by the `ESMF_Config` class, each on a new line. Lines can be continued by use of an ampersand & as a continuation symbol at the beginning of any continued line. The syntax of the problem descriptor string follows the field taxonomy syntax. The following is a basic redistribution example.

```
# Basic redistribution example
####################################
problem_descriptor_string::
'[B1G1;B2G2] --> [B1G1;B2G2]'  -d Dist.rc -g Grid.rc
'[B1G1;B2G2] --> [B1G2;B2G1]'  -d DistGrid.rc
 & otherDistGrid.rc yetanotherDistGrid.rc
 &  -g Grid.rc anotherGrid.rc
::   # end of list
```

In the above example, the two problem descriptor strings specify that a redistribution test is to be conducted, indicated by the syntax ->, between a pair of rank two blocks of memory. Following the problem descriptor string, are multiple flags and the names of specifier files. Each flag indicates a portion of the configuration space which is defined by the contents of the indicated specifier files.

argument	*definition*
-d	DELayout/DistGrid specification
-g	Grid specification

The filenames following a flag are used to specify the values for the parameter associated with that flag. The specified values from each file are combined to define that parameter's span of values. All the files associated with all the flags are combined to define the full ensemble of tests. The first problem descriptor string in the example indicates that the ensemble of distribution configurations to be tested are specified by the configurations contained within the file `Dist.rc`. Similarly, the range of grid configurations are specified within the single file `Grid.rc`. For the second problem descriptor string, the ensemble of distribution configurations to be tested are specified by the union of configurations contained within the three files `DistGrid.rc`, `otherDistGrid.rc`, and `yetanotherDistGrid.rc`. While in the first case, the range of grid configurations is specified by the single file `Grid.rc`, the second case adds the configurations found in `anotherGrid.rc` to the ensemble.

6.5.2 Problem descriptor string syntax

The problem descriptor string is contained in the problem descriptor file and describes a class of tests to be conducted. The basic syntax describes a contiguous chunk of memory spanned by some sort of logically rectangular indexing, where the left most index varies fastest in memory. Each semicolon delineated entry is associated with a native array dimension. The index locations are replaced by signifiers that express associations and distributions of the memory through the use of short descriptors.

For example $[G1\ ; G2]$ indicates that a 2D logically rectangular block of memory is associated with a 2D grid in its natural order. The signifier G represents an undistributed tensor grid. Reversing the grid signifiers $[G2; G1]$ indicates that the fastest varying dimension is instead associated with the second grid dimension. Specific information about the grid, such as its size, type, topology are left to be defined by the specifier files. It is the associations between memory and the grid that are stressed with this syntax.

To distribute the grid, a distribution signifier is needed. The block distribution of each memory dimension is indicated by the signifier B. Therefore $[B1\ G1;\ B2\ G2;\ G3]$ signifies that a 3D logically rectangular block of memory, which has a 3D associated grid, is distributed in its first two dimensions, and not its third.

Grid syntax As we have already seen, the symbol G in the problem descriptor string indicates that a dimension of a tensor grid is associated with a memory location. Alternatively, an unstructured grid is indicated with the symbol U.

As an example, consider a block of memory. To associate a tensor grid with specific dimensions of that block, the symbol G is used with a numerical suffix to indicate a specific dimension of the grid. The specific aspects of this grid are left undefined at this point, only the fact that a particular dimension of a grid is associated with a particular dimension of the memory block is implied by the grid syntax.

The complete syntax for a tensor grid specification is $Gi_g\ + H\{\# : \#\}$ where

- where i_g is the index of the grid axis,
- $H\{\# : \#\}$, is the optional signifier indicating the upper and lower sizes of the halo, where the $\#$ symbols represent integer values. The option of a separate upper and lower values allow the indication of asymmetric haloes.

The symbol to indicate a halo is appended to the grid description through use of a $+$ sign. Its absence indicates that there is no halo. Currently, only symmetric haloes are supported.

To illustrate the use of this syntax consider the following example.

```
[ G1 ; G2 + H{1:1} ]
```

This string indicates that each memory location of a 2D block of memory is associated with a grid dimension. The first memory location with the first grid dimension, and the second memory location with the second grid dimension. In addition the second memory index has an asymmetric halo of size one at the low end, and size 1 at the high end.

At times a memory location might have no grid association. The symbol $*$ is used in this case as a place holder. For example,

```
[ G2 ; G1 ; * ]
```

indicates that only the first two memory locations are associated with grid dimensions. The symbol $*$ located in the last memory location is a placeholder that indicates that location in memory has no associations. We will see later that the $*$ can be used to indicate that a memory location has both no grid and no distribution association.

In this example, the grid association has been reversed from from its natural order. The first memory location is associated with the second grid dimension, and the second memory location with the first grid dimension.

Distribution syntax The distribution syntax describes how a contiguous chunk of memory is distributed among the DEs. The syntax is analogous to the grid description syntax.Three styles of memory distribution are supported;

simple block $B\ i_D$; where i_D is the distribution space axis.

full block-cyclic $C\ i_D$; where i_D is the distribution space axis.

arbitrary A

Assuming a two dimensional distribution, the expression $[B1\ G1;\ B2\ G2]$ indicates that the two dimensional logically rectangular block of memory is;

- associated with a two dimensional grid, in its natural order.
- the first memory dimension is distributed according to the first distribution axis.
- the second memory dimension is distributed according to the second distribution axis.

The order of the distribution signifiers can be reversed just as with the grid association.

As was mentioned before, the symbol $*$ is used as a place holder to indicate a lack of association. If the third memory location of an undistributed block of memory has no grid association, it would look like this;

```
[ G1; G2; * ]
```

Alternatively if the first two dimensions of a block of memory are distributed, but all three have associated grid dimensions it would take the form;

```
[ B1 G1; B2 G2; * G3 ]
```

An example of where the third memory location would have a grid association, but no distribution, is the case where the domain has been decomposed along the horizontal, but not the vertical. This is common for a 3D field such as the ocean or atmosphere.

Lastly, if the last memory location has no associations, it would look like;

```
[ B1 G1; B2 G2; * * ]
```

The purpose of the $*$ symbols is increase readability by acting as a place holder. An illustrative example of this memory configuration is a two dimensional spatial field with multiple species or tracers represented by the third dimension.

Transformation method The action of redistribution and remapping are signified by the symbols $--{>}$ and $= \chi =>$ respectively, where χ refers to a character key representing a specific interpolation method. The currently supported interpolation methods are B for first order bilinear and P for the patch recovery method, which provides a smoother first order surface, which can meaningly differentiated. Future implementations will include C, for first order conservative, S for second order bilinear, and N for nearest neighbor. The character X is reserved for unknown or user provided methods.

χ	*Action*
B	first order bilinear interpolation
P	patch recovery interpolation
C	first order conservative interpolation
S	second order conservative interpolation
N	nearest neighbor distance weighted average interpolation
X	unknown or user provided interpolation

The problem descriptor string

```
[B1 G1; B2 G2 ] --> [B1 G1; B2 G2 ]
```

indicates an ensemble of redistribution problems where a 2D distributed source field, associated with a 2D grid, is redistributed into a new 2D *destination* distribution.

Alternatively the example

```
[ B1 G1; B2 G2 ] =C=>  [ B1 G1; B2 G2 ]
```

indicates the interpolation of one distributed 2D gridded field to another 2D grid using a first order conservative method. The interpolation method of first order conservative is indicated by placing a character C in between a pair of equal $=$ signs.

Staggering syntax (Not currently implemented) In addition to the default grid description, the stagger of the associated grid can be indicated by appending a key with an @ sign to the end of the block memory syntax. Since the stagger relates to the whole grid and not individual components of the grid, it is natural for it to be located outside of the block memory syntax $[\cdots]$.

```
[ B1 G1; B2 G2 ] @{#,#}
```

The key consists of values enclosed by a pair of brackets $\{\cdots\}$. The rank of these values, equals the rank of the associated grid. Thus in the above example, there are two values indicating the stagger since the grid is of rank two.

The stagger location key represents the location of a field with respect to the cell center location. It is indicated by relative cartesian coordinates of a unit square, cube etc. To illustrate this further, consider the example of a 2D grid. The cell is represented by a unit square with the xy axis placed at its center, with the positive x-axis oriented *East* and the positive y-axis oriented *North*. The actual values are suppressed, only the directions $+$, and 0 are used. This geometry is for reference purposes only, and does not literally represent the shape of an actual cell.

The cell center, located at the origin of the cell, is indicated by $\{0, 0\}$. The corner of the cell is indicated by $\{+, +\}$, where the ones have been dropped to just leave the plus signs. The cell face normal to the X axis (right wall) is indicated by $\{+, 0\}$, while the face normal to the Y axis (top wall) is indicated by $\{0, +\}$. This approach generalizes well to higher dimensions.

argument	*definition*
Center	{ 0,0 }
Corner	{ +,+ }
Face normal to X axis	{+,0 }
Face normal to Y axis	{0,+ }

With these four locations, it is possible to indicate the standard Arakawa grid staggerings. A key of $\{0, 0\}$ would be equivalent to an Arakawa A-grid, while a key of $\{+, +\}$ would represent an Arakawa B-grid. Components of the C and D grids would be indicated by wall positions $\{+, 0\}$ and $\{0, +\}$.

Grid Stagger	*coordinates*
A-grid	{ 0,0 }
B-grid	{ 1,1 }
C-grid & D-grid	{ 0,1 } and { 1,0}

Therefore a key of $\{0, 0\}$ in the previous example would indicate that the stagger is located at the cell center location, or the A-grid. Typically the stagger coordinates are suppressed for the A-grid.

For example, the string

```
[B1 G1; B2 G2 ] =C=> [B1 G1; B2 G2 ] @(+,+)
```

indicates that a collection of regridding tests are to be run where a block distributed two dimensional field, is interpolated from one two dimensional grid onto a second two dimensional grid, using a first order conservative method. The source field data is located at the cell centers (an A-grid stagger location) and the destination field is to be located at north east corner (a B-grid stagger location). Additional information about the pair of grids and their distributions must be specified to run an actual test.

This process generalizes to higher dimensions. Consider

```
[B1 G1; B2 G2 ; G3 ] =C=> [B1 G1; B2 G2 ; G3 ] @(+,+,+)
```

indicates that a collection of regridding tests are to be run between a 3D source grid (cell centered) and a 3D destination grid where the field stagger location is the upper corner of the 3D cell (B-grid horizontally and top of the cell vertically).

A further discussion on grid staggers can be found in section on the ESMF Grid class in the Reference manual.

6.5.3 General Data Structures

It is possible to represent more general data structures which cannot be described as simple logically rectangular blocks of memory. These embedded structures are represented by combining multiple contiguous data chunks separated by commas.

For example, in models with nested grids or which employ multi-grid methods, it is useful to have an indexed structure of logically rectangular blocks of memory, where each block is generally a different shape and size. Such a structure can be represented with parentheses as delimiters, such as

```
( tile , [ B1 G1; B2 G2 ] )
```

Here we have a collection of 2D blocks of memory, each block being associated with a 2D grid and being distributed in both dimensions.

6.5.4 Specifier files

The *problem descriptor strings* are augmented by two types of *specifier files* which complete the description of the test configuration. The specifier files indicate which members of the family described by the `problem descriptor string` are to be tested. The two types of specifier files define the grid and the distribution information needed to completely define a test. The two specifier files define information such as the type, size, and coordinates of the grid, the test function to be interpolated or redistributed, and the size of the distribution.

The nature of the grid specifier file varies depending on whether the test is a redistribution or a regridding. A redistribution test takes a field associated with a grid and rearranges it in processor space. So while a source and destination distribution are needed, only a single grid is necessary to conduct the test. A regridding test, on the other hand, takes a field associated with a source grid and interpolates it to a destination grid thaty is also part of a field. In this case both source and destination grids are needed. It is also assumed that a source and destination distribution is specified.

Redistribution	source grid source & destination distribution
Regridding	source & destination grids source & destination distribution

6.5.5 Grid Specification

For the purposes of the test harness, a grid is completely defined in terms of its type, size, coordinates, and units. From this information either a rectilinear or a curvilinear grid is generated.

The first step of the grid generation is to generate a rectilinear grid of specified size, range of coordinates, and either uniform or gaussian spacing between the coordinates. If only a rectilinear grid needed, the grid generation is finished. If a curvilinear grid is desired, the rectilinear grid is taken as a base grid and its coordinates are smoothly stretched and/or shrunk according to an analytical function, to produce a curvilinear mesh according.

Curvilinear Grid Coordinate Generation (For Future Implementation) If a curvilinear grid is desired, the rectilinear grid is taken as a base grid and its coordinates are smoothly stretched and/or shrunk according to an analytical function, to produce a curvilinear mesh according. The algebraic grid generation takes the form

$$\begin{aligned} X_{ij} &= \hat{x}_i + \epsilon \Delta x \Gamma_x(x_i, y_j) \\ Y_{ij} &= \hat{y}_j + \epsilon \Delta y \Gamma_y(x_i, y_j) \end{aligned}$$

where the new curvilinear coordinates X_{ij} and Y_{ij} are functions of the original rectilinear grid coordinates $\hat{x}_i$ and $\hat{y}_j$, a small parameter $0 < \epsilon < 1$, the smallest mesh spacings Δx and Δy, and an analytical function of the coordinates Γ. Four forms of Γ are supported, a contacted grid, an expanded grid, a symmetric sine perturbed grid, and an asymmetric sine perturbed grid.

key word	*formula* $\Gamma(x_i, y_j)$
contracting	$\Gamma_x(x_i, y_j) = -(x_i/L_x)(1 - x_i/L_x)(0.5 - x_i/L_x)(y_j/L_y)(1 - y_j/L_y)$ $\Gamma_y(x_i, y_j) = -(y_j/L_y)(1 - y_j/L_y)(0.5 - y_j/L_y)(x_i/L_x)(1 - x_i/L_x)$
expanding	$\Gamma_x(x_i, y_j) = +(x_i/L_x)(1 - x_i/L_x)(0.5 - x_i/L_x)(y_j/L_y)(1 - y_j/L_y)$ $\Gamma_y(x_i, y_j) = +(y_j/L_y)(1 - y_j/L_y)(0.5 - y_j/L_y)(x_i/L_x)(1 - x_i/L_x)$
symmetric_sin	$\Gamma_x(x_i, y_j) = \sin(4\pi xi/L_x)\cos(6\pi y_j)/L_y)$ $\Gamma_y(x_i, y_j) = \sin(4\pi xi/L_x)\cos(6\pi y_j)/L_y)$
asymmetric_sin	$\Gamma_x(x_i, y_j) = \sin(4\pi xi/L_x)\cos(6\pi y_j)/L_y)$ $\Gamma_y(x_i, y_j) = \sin(4\pi yj/L_y)\cos(6\pi x_i)/L_x)$

These four choices for Γ equate into four options for analytically generated curvilinear grids.
Examples of these four curvilinear grid options are plotted in figure 1.

This adds up to a total of 2 types of rectilinear grids (uniform and gaussian), each with three options (no connection, spherical pole connection, and a periodic option). A large variety of grids can be formed by mixing and matching the grid types and options. For example a standard latitude-longitude grid on a sphere is formed by using the pair of grid types uniform_periodic and uniform_pole. A gaussian grid is formed with uniform_periodic and gaussian_pole. And a regional grid on a sphere, without periodicity, with uniform and uniform. A summary of the rectilinear grid types is given below.

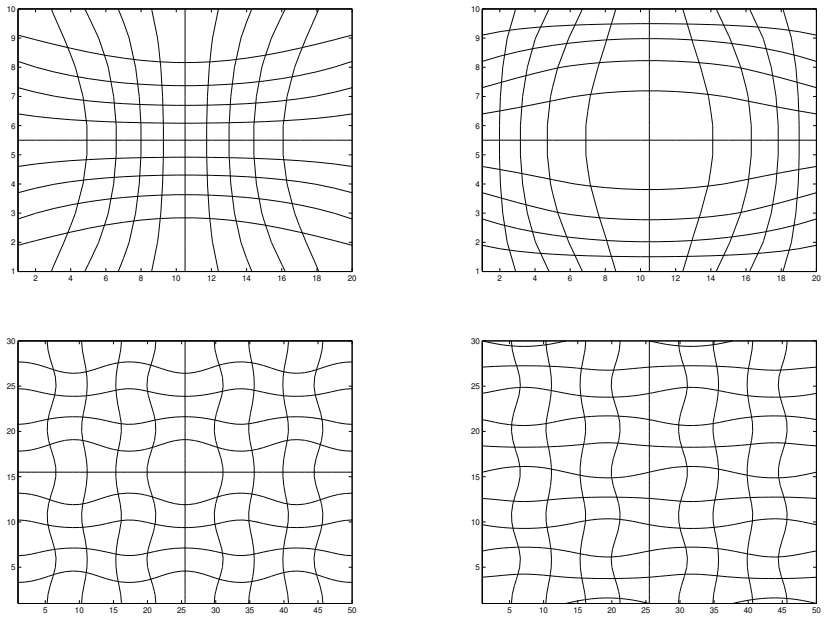

Figure 1: The four options for smooth curvilinear grids; clockwise from the top left corner, a contacted grid, an expanded grid, a symmetric sine perturbed grid, and an asymmetric sine perturbed grid.

Rectilinear grid options

grid type	*modifier*
UNIFORM	*none*
	_POLE
	_PERIODIC
GAUSSIAN	*none*
	_POLE
	_PERIODIC

The various curvilinear grid types are created in the same way of mixing and matching grid type options. A regional expanding grid is formed using the grid types expanding and expanding. Likewise a rotated regional grid is created by using the grid types rotation_15degrees and rotation_15degrees to indicate the grid type. A summary of the curvilinear grid types is given below.

Curvilinear grid options

grid type	*modifier*
CONTRACTING	*none* _POLE _PERIODIC
EXPANDING	*none* _POLE _PERIODIC
SYMMETRIC_SIN	*none* _POLE _PERIODIC
ASYMMETRIC_SIN	*none* _POLE _PERIODIC
rotation_15degrees	*none*
rotation_30degrees	*none*
rotation_30degrees	*none*

It should be noted that the rotated grid type only supports non-connected domains, and therefore has no option for _pole and _periodic connectivity.

Specifier file syntax for redistribution Since a redistribution test takes either an array or a field and rearranges it in processor space, the test requires only a single grid for the source and destination. The information required is:

parameters to define a grid for redistribution
Grid rank
Grid dimensions
Range of the coordinate axis
Coordinate units

This is the information is provided by the redistribution grid specifier file. An example of a redistribution grid specifier file is provided below.

```
# grid.rc
#########################################
map_type: REDISTRIBUTION

# regular rectilinear Grid specification
map_redist::
#rank spacing  size  range (min/max)  units
#
2  'UNIFORM'  120  -3.14159  3.14159  'RAD'   'UNIFORM'  90  -1.57  1.57  'RAD'
#
2  'UNIFORM'  400  -180   180  'DEG_E'  'GAUSSIAN'  200 -88   88   'DEG_N'
::
```

The first piece of information required for the file is the `map_type` key. It should be set to `REDISTRIBUTION` to indicate that it is intended to define grids for a redistribution test. Next is a configuration table which specifies multiple grids. The specification can be stretched over multiple table lines by use of the continuation symbol &. The order of information is as follows:

parameters to define a grid for redistribution
Grid rank
Grid axis type
Grid size
Range of the coordinate axis
Coordinate units

There are two 2D grids specified in the file. The first is a standard uniformly spaced latitude-longitude grid, where none of the grid coordinates has any topological connections. The horizontal coordinates are in radians. The second 2D grid is a gaussian spherical grid. The gaussian grid has a spherical topology and is in degrees.

Four types of coordinate units are supported; degrees, radians, meters and kilometers. Each has multiple equivalent key words.

Supported units

name	*key word*
Degrees	DEGREES DEG DEG_E DEG_N
Radians	RADIANS RAD
Meters	METERS M
Kilometers	KILOMETERS KM

Specifier file syntax for regridding The process of regridding takes a distributed test field from a source grid and interpolates it to a distributed destination grid. Therefore it requires information specifying both a source and destination grid, and an explicit test function to interpolate. The information required is:

parameters to define a grid for remapping
source and destination grid rank
source and destination grid axis type
source and destination grid dimensions
source and destination range of coordinate axis
source and destination coordinate units
test function with parameters to interpolate

This information is provided by the regridding grid specifier file. An example of a regridding grid specifier file is provided below.

```
# grid.rc
########################################
map_type: REGRID

#############################################################################
# grid | source | grid    |    grid   | grid  | units | destination |
# rank |  tag   | spacing | dimension | range |       |    tag      |
#############################################################################
# Grid specification for regridding

#rank spacing  size  range (min/max)  units
```

```
map_regrid::
# example of a pair of 2D periodic grids
2   SRC    UNIFORM_PERIODIC   120   -3.14159  3.14159  RADIANS
&              UNIFORM_POLE             90    -1.57        1.57        RADIANS
&   DST    UNIFORM_PERIODIC   120  -180      180             DEG_E
&              GAUSSIAN_POLE           88    -88        88             DEG_N
&   FUNCTION    CONSTANT   2.1 0.1  END
::
```

The first piece of information required for the file is the `map_type` key. It should be set to `REGRID` to indicate that it is intended to define grids for a regridding test. Next is a configuration table which specifies multiple pairs of grids. The specification can be stretched over multiple table lines by use of the continuation symbol &. The order of information is as follows:

parameters to define a grid for reremapping
Grid rank
Source grid axis type
Source grid size
Source range of the coordinate axis
Source coordinate units
Destination grid axis type
Destination grid size
Destination range of the coordinate axis
Destination coordinate units
Test function and parameters

The regridding grid specifier file contains a pair of 2D grids. The source grid is a standard latitude-longitude grid on a spherical topology. The destination grid is a spherical gaussian grid also on a spherical topology. The source grid is in radians, while the destination grid is in degrees. The test function is a periodic function of the grid coordinates.

The one new piece of information for the regridding specifier files are the predefined test functions. The test functions provide a physical field to be interpolated and are generated as an analytical function of the grid coordinates. Supported options include:

Test functions

name	*function*	*parameters*
constant	set value at all locations	value, relative error
coordinate	set to a multiple of the coordinate values	scale, relative error
spherical harmonic	periodic function of the grid coordinates	amplitudes and phases, relative error

The CONSTANT value test function sets the field to the value of the first parameter following the name. The second value is the relative error threshold. For example, the test function specification:

```
&   FUNCTION    CONSTANT   2.1 0.1  END
```

indicates that the field is set to the constant value of 2.1, with a relative error threshold of 0.1. This test function specification holds for grids of any rank.

The COORDINATE test function sets the field to the value of the one of the grid coordinates multiplied by the value of the first parameter following the name. The second value is the relative error threshold. For example, the test function specification:

```
&   FUNCTION    COORDINATEX   0.5 0.1  END
```

indicates that the field is set to one half of the X coordinate values. For a 2D grid the coordinate options include COORDINATEX and COORDINATEY. For a 3D grid there is the additional option of COORDINATEZ. Again, the relative error threshold is 0.1. This test function specification holds for grids of any rank.

The SPHERICAL_HARMONIC test function sets the field to the periodic harmonic function $|a+b|+a*\cos(2\pi*k_x*x/L_x)+b*\sin(2\pi*l_y*x/L_y)$. The parameters follow the order

```
&   FUNCTION    SPHERICAL_HARMONIC  a   kx  b   ly    rel_error  END
```

This test function specification is only valid for 2D grids.

6.5.6 Distribution Specification

The purpose of the distribution specification is to characterize the distribution of the grid. It differs from the grid specification in that it is designed to specify the distribution as both absolute values such as 2×3 PETs as well as in relative terms based on fractions of the total number of PETs. This second option is intended to allow the distribution space to scale with changing machine resources without having to change the distribution specification file.

Here is an example of a distribution specification file for block and block cyclic distributions.

```
################################################
# descriptive | source | source | operator | destination | dest | operator | end
# string      |  tag   |  rank  |  & value |     tag     | rank | & value  | tag
################################################

# table specifing 2D to 2D distributions
distgrid_block_2d2d::

# example with two fixed distribution sizes
 '(1,2)-->(2,1)'  'SRC' 2  'D1==' 1  'D2==' 2  'DST' 2  'D1==' 2  'D2==' 1  'END'

# example with one fixed and one variable distribution size
 '(1,n)-->(n,1)'  'SRC' 2  'D1==' 1  'D2=*' 1  'DST' 2  'D1=*' 1  'D2==' 1  'END'

# example with variable distribution sizes
 '(2n,n/2)-->(n/2,2n)'  'SRC'  2  'D1=*'  2    'D2=*'  0.5
 &                      'DST'  2  'D1=*'  0.5  'D2=*'  2     'END'

# another example with variable distribution sizes
 '(2n,n/2)-->(2n,(n/2)-1)'  'SRC' 2  'D1=*' 2  'D2=*'  0.5
 &                          'DST' 2  'D1=*' 2  'D2=*'  0.5  'D2=+' 1   'END'
::

# table specifing 3D to 3D distributions
distgrid_block_3d3d::

# example with two fixed distribution sizes
 '(1,2,1)-->(2,1,1)'  'SRC' 3  'D1==' 1  'D2==' 2  'D3==' 1
 &                    'DST' 3  'D1==' 2  'D2==' 1  'D3==' 1  'END'
::
```

The contents of the file are tables which specify pairs of distributions. Each table specifies a particular rank of distributions. The first table consists of 2D to 2D distributions, while the second table consists of a 3D to 3D pair of distributions. The order of information is as follows:

parameters to define a grid distribution
Descriptive string
Source key
Source distribution rank
Source distribution axis and size for each dimension of the distribution
Destination key
Destination distribution rank
Destination grid size
Destination distribution axis and size for each dimension of the distribution
Termination key

The first example in the table starts with a string used by the report as a brief description of the distribution. Such as $(1,2) - - > (2,1)$ which indicates a source distribution of 1×2 and a destination distribution of 2×1. Next comes a key word, either `SRC` or `DST` to indicate the beginning of the source and destination descriptions, respectively. Following each tag is a specification of the distribution. In the first case a fixed source distribution is specified by the entries $D1 == 1$ and $D2 = 2$. This indicates that the source distribution is fixed to be 1×2, provided that the test is running with at least two processors. Likewise the destination distribution is fixed to 2×1.

The second example, illustrates how to indicate a scalable distribution. Again the entry $D1 == 1$ indicates that the first dimension of the distribution is set to one, but the entry $D2 = *1$ has a different meaning. It takes the total number of PETs NPETS and scales it by one. Therefore the source distribution becomes $1 \times NPETS$. It automatically scales with the number of PETs. Likewise, the destination distribution is set to $NPETS \times 1$.

The third example is completely dynamic. Since both $D1$ and $D2$ are scalable, each dimension starts with N PETs, where N is the square root of $NPETS$ rounded down to the nearest integer. Therefore $N \times N \leq NPETS$. So if $NPETS = 4$, Then $N = 2$. If $NPETS = 6$ then N still equals 4. This base value is then modified by the indicated entries. In this case the source distribution is $2N \times 1/2N$, since the tag $D1 = *2$ indicates that first dimension is the result of the base value being multiplied by two, and the second dimension is the result of the base value being multiplied by one half. Likewise, the destination distribution is set to $1/2N \times 2N$, no matter the number of $NPETS$.

For a rank three scalable distribution, N is the cube root of NPETS rounded down to the next integer. And so on for higher rank distributions.

The fourth example illustrates the last option in the syntax. Again, since the distribution specifies two scalable distributions, N is the square root of $NPETS$ rounded down to the nearest integer. The source distribution is exactly the same as in the third example, but destination distribution has a new entry $D2 = +1$. The first dimension of the destination distribution is set to $2N$ by the entry $D1 = *2$. the second dimension is first set to $1/2N$ by the entry $D2 = *0.5$, but then modified further to $(1/2)(N) + 1$ by the entry $D2 = +1$. The resulting destination distribution is $2N \times (1/2)(N) + 1$.

The syntax for modifying the size of the distribution space combines according to the order of the operations. The entries $D2 = *0.5$ and $D2 = +1$, are not identical to $D2 = +1$ and $D2 = *0.5$, which would result in a dimension of size $(1/2)(N + 1)$.

Three operations are supported:

Distribution specification operations	
D# ==	specify a fixed value
D# =*	multiply a base value by a constant
D# =+	add a constant to the base value

6.5.7 Class Specification

The class specific specifier file is current unused, but is made available for future test expansion.

6.6 Reporting test results

The test harness offers the option of producing a human readable report on the test results. The report consists of a concise summary of the test configuration along with the test results. The test configuration is described in terms of the Field Taxonomy syntax and user provided strings. The intent is not to provide a exhaustive description of the test, but rather to provide a useful description of the failed tests.

Consider a problem descriptor string consisting of two descriptor strings describing an ensemble of remapping tests.

```
[ B1 G1; B2 G2 ] =C=> [ B1 G1; B2 G2 ] @{+,+}
[ B1 G1; B2 G2 ] =B=> [ B1 G1; B2 G2 ] @{+,+}
```

Suppose the associated specifier files indicate that the source grid is rectilinear and is 100 X 50 in size. The destination grid is also rectilinear and is 80 X 20 in size. The remapping is conducted from the A-grid position of the source grid to the B-grid stagger of the destination grid. Both grids are block distributed in two ways, 1 X NPETS and NPETS X 1. And suppose that the first dimension of both the source and destination grids are periodic. If the test succeeds for the conservative remapping, but fails for one of the first order bilinear remapping configurations, the reported results could look something like

```
SUCCESS: [B1 G1; B2 G2 ] =C=> [B1 G1; B2 G2 ] @{+,+}
FAILURE: [B1{1} G1{100}+P; B2{npets} G2{50} ] =B=>
                          [B1{1} G1{80}+P; B2{npets} G2{20} ] @{+,+}
     failure at line 101 of test.F90
SUCCESS: [ B1{npets} G1{100} +P; B2{1} G2{50} ] =B=>
                          [ B1{npets} G1{80}+P; B2{1} G2{20} ] @{+,+}
```

Notice that the problem descriptor string differs from that of the configuration files. This is because it represents specific realizations of ensemble. For example

```
[ B1{npets} G1{80}+P; B2{1} G2{20} ] @{+,+}
```

indicates that the 2D block of memory is periodic in the first dimension by the addition of the $+P$ to the first grid key. The size of the grid is indicated as 80×20 by the size arguments appended to the grid indicators $G1\{80\}$ and $G2\{20\}$. The size of the distribution is indicated in the same way as $npets \times 1$ by the block distribution indicators $B1\{npets\}$ and $B2\{1\}$.

The report indicates that all the test configurations for the conservative remapping are successful. This is indicated by the key word SUCCESS which is followed by the successful problem descriptor string. Since all of the tests in the first case pass,there is no need to include any of the specifier information. For the second ensemble of tests, one configuration passed, while the other failed. In this case, since there is a mixture of successes and failures, the report includes specifier information for all the configurations to help indicate the source of the test failure.

The supplemental information, while not a complete problem description, since it lacks items such as the physical coordinates of the grid and the nature of the test field, includes information crucial to isolating the failed test.

7 Conventions

Code and documentation developed for the ESMF will follow conventions for both style and content. The goal is to create software with a consistent look and feel so that it is easier to read and maintain.

Because much ESMF documentation is generated directly from source code, code and documentation conventions are closely intertwined.

7.1 Docs: Code and Documentation Templates and Associated Scripts

ESMF maintains a set of templates for developing Fortran and C++ codes and their documentation. The documentation templates include requirements, design, and reference documents. These templates are bundled with the source distribution, in the following directory:

```
$(ESMF_DIR)/scripts/doc_templates
```

7.1.1 Documentation Generation Script

The `do_newdoc` script in the document templates package creates a new document directory and populates it with the appropriate files. The READMEs in the package describe the procedure for running the script.

7.1.2 Code Generation Scripts

A set of scripts is available that creates a set of new directories for either a C++ or Fortran class and populates them with the appropriate code and document files. The READMEs in the `doc_templates` package describe the procedures for doing this.

The following scripts are included in the `doc_templates` package:

do_newcomp This script creates a new component. It makes a new component directory tree if one does not already exist. In addition, it adds the required template files in the component directory tree with the proper root file names.

do_newclass This script creates a new class template in the current directory. It may be used for both C++ and Fortran 90 classes. For C++, it adds an include file (ESMC_class.h) in the component include directory, if one does not exist.

7.2 Docs: Documentation Guidelines and Conventions

7.2.1 Accessibility

Software documentation for the last public release is at:

```
http://www.earthsystemmodeling.org -> Users -> Documentation
```

Software documentation for all releases is at:

```
http://www.earthsystemmodeling.org -> Download -> View All Releases
```

Documents are available to developers for editing through the ESMF repository on the SourceForge site.

7.2.2 File format

Documents will be available in both web-browsable (e.g. html) and printer-friendly formats. They will be written in LaTeX and based on a set of templates. LaTeX was chosen because:

- it is a format that many people are already familiar with;
- it can produce both print and web documentation by using text-to-html tools such as latex2html;
- it is the format generated by the F90/C++ compatible auto-documentation tool ProTeX[3], which we shall use.

7.2.3 Typeface and Diagram Conventions

The following conventions for fonts and capitalization are used in this and other ESMF documents.

Style	Meaning	Example
italics	documents	*ESMF Reference Manual*
`courier`	code fragments	`ESMF_TRUE`
`courier()`	ESMF method name	`ESMF_FieldGet()`
boldface	first definitions	An **address space** is ...
boldface	web links and tabs	**Developers** tab on the website
Capitals	ESMF class name	DataMap

ESMF class names frequently coincide with words commonly used within the Earth system domain (field, grid, component, array, etc.) The convention we adopt in this manual is that if a word is used in the context of an ESMF class name it is capitalized, and if the word is used in a more general context it remains in lower case. We would write, for example, that an ESMF Field class represents a physical field.

Diagrams are drawn using the Unified Modeling Language (UML). UML is a visual tool that can illustrate the structure of classes, define relationships between classes, and describe sequences of actions. A reader interested in more detail can refer to a text such as *The Unified Modeling Language Reference Manual.* [**?**]

7.2.4 Style Rules for LaTeX

General style recommendations for LaTeX documentation include the following:

1. Limiting the source file text width to 80 characters;
2. The use of proper indentation in sections and environments;
3. The liberal use of blank lines to delimit blocks;
4. Leaving as much of the typesetting as possible to LaTeX itself (user-specified lengths, pagebreaks, etc. are discouraged);
5. The use of the `html` package for convenient production of HTML and print formats;
6. The use of the `fontenc` packages to permit use of LaTeX special tokens in source;
7. The use of a Type-I font (e.g the `times` or `palatcm` packages) to provide scaling PDF documents.

7.3 Docs: Performance Report Conventions

Measurements and analysis of ESMF performance are documented in performance reports. These reports are used to communicate information about framework performance, and are used to define and assess the success of optimization efforts. They are posted on the ESMF website under the **Performance** link. Reports typically contain the following sections: Objective, Description of the Benchmark, Results, Optimizations and Conclusions.

Performance reports should contain the following specific information:

1. The tag or version of the framework used (*not* a moving tag, like HEAD or LAST_STABLE)
2. Where to get the source code
3. Machine location and type
4. OS and compiler version
5. Processor counts
6. Complete description of grids and distributions or where to get that info

7. How many iterations were performed for each data point

8. How the timings were obtained from the iterations

7.4 Docs: Reference Manual Conventions

7.4.1 Description, Use and Examples, and Other Introductory Sections

1. These sections shall contain thorough coverage of class behavior and usage. Any mention of private classes or implementation strategies shall be reserved for the sections called *Design and Implementation Notes*. Restrictions shall be listed in the *Restrictions and Future Work* sections.

2. If a method name is referred to in the text it shall be in this fashion `ESMF_<Class><MethodName>()`. If a code fragment or flag is included in the text, we shall use the form `ESMF_<FLAG_NAME>` or surround the fragment with the LaTeX `verbatim` command. The goal is to get all code fragments in an identifiable font.

3. Class names shall be capitalized in these sections, and in the normal text font, e.g. Field. The reason is that Field is not an ESMF derived type or code fragment and writing the full type, e.g. `ESMF_Field`, everywhere makes long explanations less readable.

4. Verbs like halo and regrid shall not be capitalized since they are not public classes. The exception is when such a verb is also a private class, and it is written about in that context. For example, when writing about Regrid as a private class in a *Design and Implementation Notes* section it is appropriate to capitalize it.

5. Full class names shall be used everywhere (e.g., Layout will not be used as a shorthand for DELayout).

7.4.2 Examples Sections

A standard template for examples has not yet been created.

7.4.3 Flags and Options Sections

1. All flags and options (hereafter referred to as options) shall be broken out into separate sections called <*Class*> *Options*. Options shall each have their own derived type, e.g. `ESMF_Sync_Flag`, and valid values will be all capitals with underscore separators, e.g. `ESMF_SYNC_NONBLOCKING`.

2. When possible, call options <Class>Type or <Something>Flag and name the argument in the argument list the same thing. For example, the option `ESMF_Sync_Flag` has associated arguments called `syncflag`.

3. The purpose of options shall be fully described along with all valid values in the options sections of the document. See the standard format in *Calendar Options* or *Grid Options*.

4. In the sections describing the usage and values for options, only use the full class name, e.g. `ESMF_Field`, method names such as `ESMF_FieldWrite`, or particular argument names, such as `syncflag`. Do not use the more informal form of the short class name in a regular font, e.g. Field. The rationale is that here we are carefully specifying interface behavior and are referring only to particular ESMF types, values, and methods. It is a pain in the neck to be that rigorous in the longer text sections (like *Description*), but it is helpful to be precise for the API specification.

7.4.4 Class API Sections

1. Public class methods shall be capitalized by name within a file. Generally private methods are listed at the end of the file. Especially for classes with many methods, this makes locating a particular method somewhat easier.

2. All public methods shall be fully documented in the *Reference Manual*. Private methods without interfaces shall not appear at all. Private methods that are overloaded with public interface shall follow a standard format that clearly indicates that the methods are overloaded; see `ESMF_FieldCreate()` for an example.

3. The *Reference Manual* shall include for each class a thorough specification of behavior, including any default values or behaviors. Description of options can reference *<Class> Options* sections. All descriptions should contain complete sentences with periods, e.g. "This method sets invalid values for an `ESMF_FieldBundle` object."

4. The brief one-line description that follows the method name shall begin with a capital and have no period at the end.

5. All methods shall conform to method naming conventions.

6. Functions shall have their data type and return variable name declared on a separate line after the `function` statement, and prior to the argument list declarations. The data type shall not appear in the `function` statement. The optional `result` clause should also not be used.

7. All arguments shall conform to variable naming conventions for capitalization and standard use of terms such as count, dim, etc. The conventions shall be used even if names become fairly long. All class names will be used in full, e.g. DELayout will not be used instead of DELayout. Cryptic abbreviations like `btype` shall not be used - use `bundleType` instead.

8. As with descriptions of flags and options: in the sections describing methods, only use the full class name, e.g. `ESMF_Field`, method names such as `ESMF_FieldWrite`, or particular argument names, such as `syncflag`. Do not use the more informal short form in regular font, e.g. Field.

7.5 Code: Method Conventions

7.5.1 Standard Method Names

ESMF defines a set of standard methods and interface rules that hold across the entire API. These are:

- `ESMF/C_<Class>Create()` and `ESMF_<Class>Destroy()`, for creating and destroying classes. The `ESMF_<Class>Create()` method allocates memory for the class structure itself and for internal variables, and initializes variables as appropriate. It is always written as a Fortran function that returns a derived type instance of the class.

- `ESMF/C_<Class>Set()` and `ESMF_<Class>Get()`, for setting and retrieving a particular item or flag. In general, these methods are overloaded for all cases where the item can be manipulated as a name/value pair. If identifying the item requires more than a name, or if the class is of sufficient complexity that overloading in this way would result in an overwhelming number of options, we define specific `ESMF_<Class>Set<Something>` and `ESMF_<Class>Get<Something>` interfaces.

- `ESMF/C_<Class>Add()`, `ESMF_<Class>Get()`, and `ESMF_<Class>Remove()` for manipulating items that can be appended or inserted into a list of like items within a class. For example, the `ESMF_StateAdd()` method adds another Field to the list of Fields contained in the State class.

- `ESMF/C_<Class>Print()`, for printing the contents of a class to standard out. This method is mainly intended for debugging.

- `ESMF/C_<Class>ReadRestart()` and `ESMF_<Class>WriteRestart`, for saving the contents of a class and restoring it exactly. Read and write restart methods have not yet been implemented for most ESMF classes, so where necessary the user needs to write restart values themselves.

- ESMF/C_<Class>Validate(), for determining whether a class is internally consistent. For example, ESMF_FieldValidate checks whether the Array and Grid associated with a Field are consistent.
- ESM[F/C]_<Class>Construct() and ESM[F/C]_<Class>Destruct(), for initializing a previously allocated object with valid data. This function is called by the create function. Depending on the type of object this function may or may not allocate resources that need to be freed. Only for ESMF internal usage; not visible to users.

7.5.2 Use of *Set and *Get

In general, Set and Get calls with optional arguments for each possible attribute to be processed shall be used instead of individual Set and Get calls. The goals are a smaller, clearer interface and easier extensibility. An exception is when the Set or Get does not translate to a simple name/value pair, for instance when one element of an array of internal attributes must be retrieved.

7.5.3 Use of Is* and Has*

Methods names that indicate the presence of some attribute or quality shall use the Has* form. Methods that indicate a condition shall use the Is* form; for example, ESMF_DELayoutIsLocal. In Fortran, these should return logical data type results. The goals are consistency and clarity.

7.5.4 Functions vs. Subroutines

Most Fortran calls in the ESMF are subroutines, with any returned values passed through the interface. For the sake of convenience, some ESMF calls are written as functions.

7.5.5 Source and Destination Ordering

In ESMF we shall follow a source first, then destination convention when ordering arguments.

7.6 Code: Argument Conventions

The naming conventions listed here apply to arguments that appear in ESMF public interfaces. It is nice if internal arguments also adhere to these conventions, but not enforced as it is as the interface.

7.6.1 Standard Variable Names

The following table lists a set of standard terms that we use in ESMF.

Cell	grid cell
Coord	coordinate
Count	count
Dim	dimension, used for grids
Dst	destination
List	indicates that the quantity is an array
Local	indicates that the quantity is associated with a PET or DE
Per	per item
Rank	data array dimension
Send	send
Src	source
Recv	receive

These are used in combination to create argument and variable names. For example, localCellCountPerDimList is an array of counts, per dimension, of the number of local grid cells.

7.6.2 Use of Is* and Has*

Variable names that indicate the presence of some attribute or quality shall use the Has* form; e.g., a variable such as `hasGrid`. Variables that indicate a condition shall use the Is* form. In Fortran, these should return `logical` data type results. The goals are consistency and clarity.

7.6.3 Variable Capitalization

For method arguments that are multi-word, the first word is lower case and subsequent words begin with upper case. ESMF class names (including typed flags) are an exception. When multi-word class names appear in argument lists, all letters after the first are lower case. The first letter is lower case if the class is the first word in the argument and upper case otherwise. For example, in an argument list the DELayout class name may appear as `delayout` or `srcDelayout`.

7.6.4 Variables Associated with Options

Variables associated with flags or option shall have the same name as the option (as long as it is not too awkward); for example, an option `ESMF_Sync_Flag` has an associated argument `syncflag`.

7.6.5 Variables Having Logical Data Type

Variables associated with flags or other boolean values shall use the Fortran `logical` data type in the public ESMF API. The `ESMF_Logical` derived type shall only be used in Fortran code internally, for example when passing logical values between Fortran and C++ code.

7.6.6 Arguments which are Arrays

Arguments which are arrays shall use the Fortran assumed-shape passing style. This allows the procedure to use intrinsic functions such as `size`, `lbound`, and `ubound` within the method to determine array dimension sizes. Also, ESMF convention is to place the array specification following the variable name, rather than by using the `dimension` attribute or a separate `dimension` statement. For example:

```
type(ESMF_Field), intent(in) :: fieldList(:)
```

7.6.7 Arguments which are Pointers

Arguments which are pointers may point to either scalar or array targets. Pointers are typically used when an object must be allocated within the method and returned to the caller. A second use is when the original array bounds must be maintained. No `intent` attributes are currently used with pointers because they are not part of Fortran-90 or Fortran-95.

7.7 Code: File Rules

7.7.1 Version Identification

The first line of every file in ESMF must be its CVS version identification. For example, each LaTeX file must begin:

```
% $Id$
```

In addition, for source code, the CVS identifier must have the ability to be written to a variable for output to a configuration log. Therefore each F90 module must contain a declaration:

```
character(len=80), private :: version = '$Id$'
```

7.7.2 License and Copyright Information

Source and documentation files must contain the ESMF license and copyright header:

```
! Earth System Modeling Framework
! Copyright 2002-2020, University Corporation for Atmospheric Research,
! Massachusetts Institute of Technology, Geophysical Fluid Dynamics
! Laboratory, University of Michigan, National Centers for Environmental
! Prediction, Los Alamos National Laboratory, Argonne National Laboratory,
! NASA Goddard Space Flight Center.
! Licensed under the University of Illinois-NCSA License.
```

7.7.3 TODO: Reminder

To identify code sections that need additional work or other pending activity that must be done before a release, the code will be marked with a `TODO` identifier.

```
example:
! TODO: add support for other data types (CMD 4/01).
```

This will allow developers to 'grep' source files before software releases.

7.7.4 FILENAME Macros

Source code files should define the /tt ESMF_FILENAME or /tt ESMC_FILENAME macros to ensure that the filename is correctly displayed from the ESMF logging. This can be done using the following code:

```
#define ESMC_FILENAME "./src/Infrastructure/Attribute/src/ESMCI_Attribute.C"
```

with the appropriate filename. Note that the path is relative to the ESMF root directory.

7.8 Code: Style Rules for Source Code

General style recommendations for source code include the following:

1. Limiting the text width to 80 characters;
2. Each class implementation will be in a single file;
3. The use of proper indentation of loops and blocks;
4. The liberal use of blank lines to delimit code blocks;
5. The use of comment lines of dashes or dots to delimit procedures;
6. The use of useful descriptive names for physically meaningful variables; short conventional names for iterators (e.g `(i,j,k)` for spatial grid indices);
7. The use of short comments on the same line to identify variables (i.e using `!` or `//`); longer comments in well-delineated blocks to describe what a portion of code is doing;
8. Compact code units: long procedures should be split up if possible. 200 lines is a rule-of-thumb procedure length limit.
9. Use of global variables or public class variables should be kept to a minimum to increase encapsulation and ensure modular code.

7.9 Code: Error Handling Conventions

7.9.1 Objectives

The objectives of these conventions are:

1. to produce appropriate error messages when users try to call a method or branch in the ESMF software that is incomplete;
2. to produce appropriate error messages for methods that have been implemented and reviewed, and to identify these methods clearly in the ESMF source code;
3. to create defaults that enable developers to generate appropriate error messages throughout the development process, with a minimum of effort;
4. to flexibly move between a mode where incomplete code is treated as an error (the user perspective) to a mode where it is not (the developer perspective, when using stubs for code construction).

7.9.2 Approach

ESMF methods are classified into two categories: pre-review, and post-review. As of the version 3.0.1 release, *all* methods are pre-review.

Reviews are expected to begin once conventions for framework behavior are sufficiently well-defined. To pass review, methods must have their basic functionality fully implemented. If any unimplemented options are present, they must be documented and must generate appropriate errors in order for the method to pass review.

A pre-review method has its return code (`rc`) initialized to the value `ESMF_RC_NOT_IMPL` (not implemented). Possible sources of specific errors within the method, such as subroutine calls, generate local return codes (`localrc`).

The `localrc` values are evaluated using the function `ESMF_LogMsgFoundError()`, which takes both the `localrc` and the `rc` as input arguments. If the `localrc` does not indicate an error, `rc` is output from `ESMF_LogMsgFoundError(` with the value it had going into the call, and the value of the function is `.false.`. If `localrc` does indicate an error, `rc` is set to the `localrc` value before being output from `ESMF_LogMsgFoundError()`, and the value of the function is `.true.`.

The convention is to place `ESMF_LogMsgFoundError()` into a conditional statement that causes a return from the method, with the correctly set `rc`, when an error is found in `localrc`, e.g.:

```
if (ESMF_LogFoundError(localrc, <other args ...>, rc)) return
```

Use of the `ESMF_LogMsgFoundError()` function is further described in the ESMF Reference Manual.

Note that if no error is found, `rc` retains its `ESMF_RC_NOT_IMPL` status throughout the method. The developer must explicitly set the `rc` to `ESMF_SUCCESS` just before returning from any branch that is believed to be successful.

The default `rc` value thus becomes `ESMF_RC_NOT_IMPL` for any code that is not explicitly marked as reaching success by the developer. Stub methods and incomplete branches can be handled very naturally this way - the developer simply does not set `rc=ESMF_SUCCESS` before returning. There are two differences in the treatment of pre-review and post-review methods:

1. Post-review methods have their `rc` initialized to the `ESMF_RC_NOT_SET` value. The rationale is that methods that are supposed to be implemented should not default to an error code that says that they are unimplemented. The difference in initialization also indicates a particular method's review status to someone browsing the source code.
2. Any unimplemented or incomplete branch must have its `rc` value explicitly set by the developer to `ESMF_RC_NOT_IMPL` before returning. This is to ensure that the behavior of the method is communicated as accurately as possible to the user via its return codes.

These conventions achieve the first three objectives above, namely producing appropriate error messages throughout a method's life cycle, with minimal developer effort.

7.9.3 Error Masking

Under normal use - what a user will encounter by default - ESMF_RC_NOT_IMPL is treated as an error. However, through a special LogErr setting a developer who wishes to use stub methods and prototyping during code construction can equate ESMF_RC_NOT_IMPL with ESMF_SUCCESS. This achieves the fourth objective above, allowing for different user and developer modes of handling incomplete code. This is done by setting the errorMask argument in the ESMF_LogSet() call to ESMF_RC_NOT_IMPL, e.g.:

```
call ESMF_LogSet(errorMask = /ESMF_RC_NOT_IMPL/)
```

7.9.4 Example (pre-review method)

The following is an example of a pre-review method, sub(), that calls two subroutines internally, subsub1() and subsub2(). The subroutine sub() takes as input an integer argument that it can branch on and outputs a return code. Here branch==1 is fully implemented, branch==2 is incomplete, and other values of branch are not yet addressed. Several possible error scenarios are described following the code listing.

```
subroutine sub(branch, rc)
integer :: branch
integer, optional :: rc
integer :: localrc   ! local return code

if (present(rc)) rc=ESMF_RC_NOT_IMPL
! ...code...
if (branch==1) then
call subsub1(localrc)
if (ESMF_LogFoundError(localrc, &
  ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rc=rc)) then
  !   ... any necessary garbage collection ...
  return ! Return point 1
endif !   ...fully implemented code...
if (present(rc)) rc=ESMF_SUCCESS
elseif (branch==2) then call subsub2(localrc)
if (ESMF_LogFoundError(localrc, &
  ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rc=rc)) then
  !   ... any necessary garbage collection ...
  return ! Return point 2
endif
!   ...incomplete code...
end if

end subroutine
```

Note: This example is quite artificial, since simple branching like this would be better handled with a switch case statement that had a well-defined default. However, the control sequence shown better illustrates the sorts of errors that can occur in a real code with complex conditionals.

Possible scenarios:

1. branch==1 and there is no error in subsub1(). In this case, rc retains the value of ESMF_RC_NOT_IMPL up to the point at which it is set to ESMF_SUCCESS, after which sub() ends.

2. branch==1 and there is an error in subsub1(). The localrc returned by subsub1() will have the value of a specific error code, say ESMF_RC_DIV_ZERO. This will get passed into the evaluation ESMF_LogMsgFoundErr(), which will set rc to the localrc error value. Since the value of the evaluation expression is true, the method will return at Return point 1. It is important to perform any necessary garbage collection before returning to avoid memory leaks.

3. branch==2 and there is no error in subsub2(). Here the value of rc is still ESMF_RC_NOT_IMPL when it is input into the ESMF_LogMsgFoundErr() call following subsub2(). Since the code in this branch is not complete and rc is never set to ESMF_SUCCESS, the rc value is still ESMF_RC_NOT_IMPL at the end of sub(). Note that to allow the default to work properly, rc should not be set to ESMF_SUCCESS right before the end of the method!

4. branch==2 and there is an error in subsub2(). Much like scenario 2, the localrc returned by subsub2() will have the value of a specific error code. The localrc will get passed into ESMF_LogMsgFoundErr(), which will set rc to the localrc error value, and then cause a return at Return point 2.

5. any value where branch $/=$ 1 and branch $/=$ 2 (e.g. branch==0). Here there is a "hole" in the code, since the behavior of branch==0 is not yet addressed. However, with this approach the rc value remains ESMF_RC_NOT_IMPL when sub() ends, producing an appropriate error message for the user.

7.9.5 Example (post-review method)

This example is modeled on the pre-review example, but illustrates error handling for a post-review method. Here branch==1 is fully implemented, branch==2 is known to be incomplete, and other values of branch are cases that the developer and reviewers have missed. This is not what one would wish to see in reviewed code - ideally all cases should be implemented - but we cannot count on this. This example shows how deficiencies can most effectively be handled.

The return code rc is now initialized to ESMF_RC_NOT_SET. The developer has recognized that the branch==2 code is incomplete and rc is set explicitly to ESMF_RC_NOT_IMPL before returning.

```
subroutine sub(branch, rc)
integer :: branch
integer, optional :: rc
integer :: localrc   ! local return code

if (present(rc)) rc=ESMF_RC_NOT_SET
! ...code...
if (branch==1) then
call subsub1(localrc)
if (ESMF_LogFoundError(localrc, &
  ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rc=rc)) then
  !   ... any necessary garbage collection ...      return ! Return point 1
endif
!   ...fully implemented code...
if (present(rc)) rc=ESMF_SUCCESS
elseif (branch==2) then call subsub2(localrc)
if (ESMF_LogFoundError(localrc, &
  ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rc=rc)) then
  !   ... any necessary garbage collection ...
```

```
    return ! Return point 2
  endif
  !   ...incomplete code...
  if (present(rc)) rc=ESMF_RC_NOT_IMPL       end if
  end subroutine
  {
```

Possible scenarios:

6. `branch==1`. Here the behavior is essentially the same as in scenarios 1 and 2. When there is an error in `subsub1()`, the returned `rc` from `sub()` has the error code `subsub1()` generated. When there is not an error in `subsub1()`, the returned `rc` from `sub()` is `ESMF_SUCCESS`.

7. `branch==2` and there is an error in `subsub2()`. The behavior here is analogous to scenario 3. The returned `rc` from `sub()` has the error code `subsub2()` generated.

8. `branch==2` and there is no error in `subsub2()`. The value of `rc` remains `ESMF_RC_NOT_SET` up to the point at which it is explicitly set to `ESMF_RC_NOT_IMPL`, and `sub()` ends. Method documentation is expected to explain the gap in functionality.

9. any value where branch $/=1$ and branch $/=2$ (e.g. `branch==0`). Here `rc` is not explicitly set to a value before the branch returns, and so `sub()` returns with the default `rc` value of `ESMF_RC_NOT_SET`. The user can tell from this error code that the developer did not consider this either unimplemented code or successful code, and that there is an unexpected "hole" in the method's functionality.

7.9.6 Memory Allocation Checking

In Fortran code, the `allocate` and `deallocate` statements should always have their success or failure checked by using `stat=` and checking the value it returns. Since this value is defined by the Fortran standard, and is not an ESMF return code, the alternative checking calls are `ESMF_LogFoundAllocError()` and `ESMF_LogFoundDeAllocError()`. To further distinguish the difference between a Fortran status return and a ESMF return code, the local error variable should be named "memstat" rather than "localrc".

```
real, allocatable :: big_array(:,:)
integer :: memstat
! ...
allocate (big_array(m,n), stat=memstat)
if (ESMF_LogFoundAllocError(memstat, &
  ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rc=rc)) then
  ! ... any necessary clean up
  return
end if
! ... use big_array for a while
deallocate (big_array, stat=memstat)
if (ESMF_LogFoundDeAllocError(memstat, &
  ESMF_ERR_PASSTHRU, &
  ESMF_CONTEXT, rc=rc)) then
  ! ... any necessary clean up
  return
end if
```

In C++ code, where the `new` operator is used, allocation errors can be caught by wrapping the allocation code inside a `try` block to catch any exceptions. Multiple allocations may be wrapped inside a single block in order to reduce the amount of checking code.

```
#include "ESMCI.h"
//...
void alloc_func (int m, int n, int *rc) {
#undef  ESMC_METHOD
#define ESMC_METHOD alloc_func
  double *array1, *array2;
  try {
    array1 = new double[m];
    array2 = new double[n];
  }
  catch (std::bad_alloc) {
    // ... any necessary clean up
    ESMC_LogDefault.MsgAllocError("allocating array1 and array2",
        ESMC_CONTEXT, rc);
    return;
  }
  // ... use array1 and array2 for a while
  delete array2;
  delete array1;
}
```

In code which uses `malloc()` and `free()`, the return value from `malloc()` may be tested against `NULL` to determine if an error occurred. No checking is possible with `free()`. In new code, the C++ `new` operator and associated checking should be used.

7.10 Initialization Standardization Instructions

7.10.1 Overview

Currently not all Fortran compilers support the automatic initialization of class components. However, in the ESMF system this initialization is used for two tasks. The first is the initialization of shallow class components which need to be set before they're used (e.g. flags). The second is the detection of invalid (e.g. not yet created) deep class variables. In order to handle these tasks in the absence of compiler initialization a software based solution has been developed.

The software solution is based on the improbability of an uninitialized variable containing a specific value. For example, given a 64 bit integer variable x, the probability of x containing the value 10 are approximately 1 in 10^{19}. This of course assumes that the value in an uninitialized variable is a uniformly distributed random variable, which is not necessarily true. However, the small probability of a given value occurring is not an unreasonable assumption given that the value is not pathological (e.g. 0 or binary 111111..). The probability can also be made as small as is necessary by using larger precision types.

Given the improbability of a specific value occurring in an uninitialized variable, the task of initializing a shallow class is implemented as follows. First, a non-pathological value, ESMF_INIT_DEFINED, is chosen. Next, an integer component, isInit, is added to the shallow class. Finally, for every routine taking the shallow class as an input variable, the component isInit is checked against ESMF_INIT_DEFINED. If they don't match, then the variable containing isInit is initialized, including setting isInit to ESMF_INIT_DEFINED. If they do match, then with extremely high probability the variable has already been initialized and nothing need be done. This algorithm helps to ensure that the shallow class is initialized before being used in the system.

The task of detecting an invalid deep class is implemented as follows. As before, we chose a value ESMF_INIT_CREATED, and add the isInit component to the class definition. Next, inside the constructors for the class isInit is set to

ESMF_INIT_CREATED, and inside the destructors for the class isInit is set to something other than ESMF_INIT_CREATED. Finally, for every routine taking the deep class as an input variable, the component isInit is checked against ESMF_INIT_CREATED. If they match, then nothing happens because with extremely high probability the variable has been though the constructor and thus is valid. If they don't match, then the variable is invalid and an error occurs. This allows invalid deep class variables to be detected before being used in the system.

In order to make the process of adding these non-compiler based initialization tasks easier, a set of macros has been defined. The following is a description of the procedures to use them.

7.10.2 Instructions

These changes should be added to all new F90 code in both the source (/src) and interface (/interface) subdirectories of the classes.

When adding code as a part of this task remember to use the standard ESMF coding conventions. In particular, when adding new routines remember to use the standard ESMF protex style (BOP/EOP) subroutine description, like those that appear in every other ESMF subroutine. For Validate use the BOP/EOP tags for ESMF_TYPEGetInit and ESMF_TYPEInit use the internal BOPI/EOPI tags.

Some F90 source files are autogenerated. In this case, modify the macro file which is used to create the autogenerated file. The macro file is usually named something like ESMF_TypeMacos.h. When modifying a macro file remember to end each line with: @ To recognize when an F90 file is autogenerated they usually have the line: <Created by maco - do not edit directly> close to the beginning. There will also usually be a .cpp file with the same name as the .F90 file.

For the purposes of initialization standardization the classes in the system have been divided into three types: shallow, deep, and parameter. Shallow classes have little if any allocation and can often be used right after declaration. An example of a shallow class is ESMF_ArraySpec. Deep classes on the other hand have a lot of allocated data and need to be constructed before being used. An example of a deep class is ESMF_State. Parameter classes are those which are used to add type checking to a set of parameter values. This type of class is typically just a single integer wrapped in a Fortran type. An example of a parameter class is ESMF_GridType. Both shallow and deep classes are effected by the standardization, but parameter classes are left unchanged. See section 7.10.8 for a list of ESMF classes and their classification.

The instructions for the standardization have been broken up by what to do for each module, what to do for each type of class, and what to do for routines.

7.10.3 Module

When adding a new F90 module do the following:

1. If the following line is not present in the use statement list, then add it (except in ESMF_LogErr.F90 and ESMF_UtilTypes.F90):

```
use ESMF_InitMacrosMod
```

2. If the following line is not present in the include statement list, then add it (except in ESMF_LogErr.F90):

```
#include "ESMF.h"
```

7.10.4 Shallow Class

When adding a new shallow class definition perform the following steps:

1. If the line:

```
use ESMF_UtilTypesMod
```

isn't in the list of used modules, then add it.

2. Add the line ESMF_INIT_DECLARE to the type definition.

3. Be sure the #ifndef ESMF_NO_INITIALIZERS construct is used around any initialization of type components. The declarations inside the #else to #endif half of the construct should be identical to the #ifndef half except without initialization.

4. Create an initialization subroutine for the class. Given that the name of the class is ESMF_STYPE, the name of the subroutine should be ESMF_STYPEInit(). This subroutine should take a single parameter of class ESMF_STYPE and set all the variables that are initialized in the #ifndef ESMF_NO_INITIALIZERS branch of 3. Note, for pointers that are set using ptr->Null() in the type initialization, use nullify(t%ptr) in the ESMF_STYPEInit subroutine instead.

5. Add the line ESMF_INIT_SET_DEFINED(S1) to the body of the new subroutine ESMF_STYPEInit() described in 4. S1 is the variable being initialized in ESMF_STYPEInit().

6. Create an access subroutine for the new init component. The name of the subroutine should be ESMF_STYPEGetInit. It should have the following form:

```
function ESMF_STYPEGetInit(s)
   type(ESMF_STYPE), intent(in), optional :: s
   ESMF_INIT_TYPE :: ESMF_STYPEGetInit

  if (present(s)) then
      ESMF_STYPEGetInit=ESMF_INIT_GET(s)
  else
      ESMF_STYPEGetInit=ESMF_INIT_DEFINED
  endif

end function ESMF_STYPEGetInit
```

7. Create an ESMF_STYPEValidate subroutine with the following form:

```
! Comments describing validate (see ESMF_StateValidate for an example)
subroutine ESMF_STYPEValidate(s,rc)
   type(ESMF_STYPE), intent(inout) :: s
   integer, intent(out),optional :: rc

   ! check initialization status
   ESMF_INIT_CHECK_SET_SHALLOW(ESMF_STYPEGetInit,ESMF_STYPEInit,s)

   ! return success
   if (present(rc)) then
      rc=ESMF_SUCCESS
   endif

end subroutine ESMF_STYPEValidate
```

8. If the shallow class in question is public, then add the lines:

```
   public ESMF_STYPEInit
   public ESMF_STYPEGetInit
   public ESMF_STYPEValidate
```

to the PUBLIC MEMBER FUNCTION section.

9. Add the new class to the list in section 7.10.8.

Here is an example illustrating the whole procedure:

Starting with this shallow class definition:

```
   module ESMF_ExampleMod

       type ESMF_Shallow
           private
            ... other components ...
#ifndef ESMF_NO_INITIALIZERS
            integer :: num=0
            integer, pointer :: list(:) => Null()
#else
            integer :: num
            integer, pointer :: list(:)
#endif
       end type

! !PUBLIC TYPES:
      public ESMF_Shallow

! !PUBLIC MEMBER FUNCTIONS:

       contains

       ... other routines ...

   end module ESMF_ExampleMod
```

The standardization procedure yields this:
(modified lines marked with *)

```
   module ESMF_ExampleMod
*       use ESMF_UtilTypesMod
*       use ESMF_InitMacrosMod

       type ESMF_Shallow
            private
            ... other components ...
#ifndef ESMF_NO_INITIALIZERS
            integer :: num=0
            integer, pointer :: list(:) => Null()
#else
            integer :: num
            integer, pointer :: list(:)
#endif
*            ESMF_INIT_DECLARE
       end type
```

```
! !PUBLIC TYPES:
      public ESMF_Shallow

! !PUBLIC MEMBER FUNCTIONS:
*      public ESMF_ShallowInit
*      public ESMF_ShallowValidate
*      public ESMF_ShallowGetInit

       contains

*    function ESMF_ShallowGetInit(s)
*       type(ESMF_Shallow), intent(in), optional :: s
*       ESMF_INIT_TYPE :: ESMF_ShallowGetInit
*
*      if (present(s)) then
*          ESMF_ShallowGetInit=ESMF_INIT_GET(s)
*      else
*          ESMF_ShallowGetInit=ESMF_INIT_DEFINED
*      endif
*
*    end function ESMF_ShallowGetInit
*
*    subroutine ESMF_ShallowInit(s)
*       type(ESMF_Shallow) :: s
*
*        s%num=0
*        nullify(s%list)
*
*        ESMF_INIT_SET_DEFINED(s)
*    end subroutine ESMF_ShallowInit
*
*    subroutine ESMF_ShallowValidate(s,rc)
*       type(ESMF_Shallow), intent(inout) :: s
*       integer, intent(out),optional :: rc
*
*       ! check initialization status
*       ESMF_INIT_CHECK_SET_SHALLOW(ESMF_ShallowGetInit,ESMF_ShallowInit,s)
*
*       ! return success
*       if (present(rc)) then
*          rc=ESMF_SUCCESS
*       endif
*
*    end subroutine ESMF_ShallowValidate

       ... other routines ...

   end module ESMF_Shallow
```

7.10.5 Deep Class

When adding a new deep class definition perform the following steps:

1. If the line:

   ```
   use ESMF_UtilTypesMod
   ```

   isn't in the list of used modules, then add it.

2. Add the line ESMF_INIT_DECLARE to the type definition.

3. Add the line ESMF_INIT_SET_CREATED(D1) to the TypeCreate() function. D1 is the deep class variable being created.

4. Add the line ESMF_INIT_SET_DELETED(D1) to the TypeDestroy() subroutine. D1 is the deep class variable being destroyed.

5. Create an access subroutine for the new init component. The name of the subroutine should be ESMF_DTYPEGetInit. It should have the following form:

   ```
   function ESMF_DTYPEGetInit(d)
      type(ESMF_DTYPE), intent(in),optional :: d
      ESMF_INIT_TYPE :: ESMF_DTYPEGetInit

     if (present(d)) then
         ESMF_DTYPEGetInit=ESMF_INIT_GET(d)
     else
         ESMF_DTYPEGetInit=ESMF_INIT_CREATED
     endif

   end function ESMF_DTYPEGetInit
   ```

6. Given that the deep class is ESMF_DTYPE create a validate subroutine with the following form:

   ```
   ! Comments describing validate (see ESMF_StateValidate for an example)
   subroutine ESMF_DTYPEValidate(d1,rc)
      type(ESMF_DTYPE), intent(in) :: d1
      integer, intent(out),optional :: rc

      ! Check Init Status
      ESMF_INIT_CHECK_DEEP(ESMF_DTYPEGetInit,d1,rc)

      ! Add other checks here

      ! If all checks passed return success
      if (present(rc)) then
          rc=ESMF_SUCCESS
      endif

   end subroutine ESMF_DTYPEValidate
   ```

7. If the deep class in question is public, then add the lines:

```
          public ESMF_DTYPEGetInit
          public ESMF_DTYPEValidate
```

to the PUBLIC MEMBER FUNCTION section.

8. Some deep classes are private and occasionally it will become necessary to access their internal components to set the initialization flag or get the this pointer in code which doesn't have access. The standard way to do this in ESMF is to write a public subroutine in the class module where the subroutine can have access to the classes' internal components.

   The following are the standard names for the subroutines which set a class as created, set a class as deleted, and set the this pointer, and get the this pointer:

   ```
   ESMF_DTYPESetInitCreated
   ESMF_DTYPESetInitDeleted
   ESMF_DTYPESetThis
   ESMF_DTYPEGetThis
   ```

9. Add the new class to the list in section 7.10.8.

Here is an example illustrating the whole procedure:

Starting with this deep type definition:

```
 module ESMF_ExampleMod

      type ESMF_Deep
           private
           ... other components ...
      end type

! !PUBLIC TYPES:
     public ESMF_Deep

! !PUBLIC MEMBER FUNCTIONS:

      contains

       function ESMF_DeepCreate()
             type(ESMF_Deep) :: ESMF_DeepCreate

             ... other create code ...

       end function ESMF_DeepCreate

       subroutine ESMF_DeepDestroy(d)
             type(ESMF_Deep) :: d

             ... other create code ...

       end subroutine ESMF_DeepDestroy

       ... other routines ...
```

```
   end module ESMF_ExampleMod
```

The standardization procedure yields this:
(modified lines marked with *)

```
   module ESMF_ExampleMod
*       use ESMF_UtilTypesMod
*       use ESMF_InitMacrosMod

       type ESMF_Deep
           private
           ... other components ...
*           ESMF_INIT_DECLARE
       end type

! !PUBLIC TYPES:
      public ESMF_Deep

! !PUBLIC MEMBER FUNCTIONS:
*      public ESMF_DeepValidate
*      public ESMF_DeepGetInit

       contains

*    function ESMF_DeepGetInit(d)
*       type(ESMF_Deep), intent(in),optional :: d
*       ESMF_INIT_TYPE :: ESMF_DeepGetInit
*
*       if (present(d)) then
*          ESMF_DeepGetInit=ESMF_INIT_GET(d)
*      else
*          ESMF_DeepGetInit=ESMF_INIT_CREATED
*      endif
*
*    end function ESMF_DeepGetInit
*
*       subroutine ESMF_DeepValidate(d,rc)
*          type(ESMF_Deep), intent(in) :: d
*          integer, intent(out),optional :: rc
*
*          ! Check Init Status
*          ESMF_INIT_CHECK_DEEP(ESMF_DeepGetInit,d,rc)
*
*         ! Add other checks here
*
*         ! If all checks passed return success
*         if (present(rc)) then
*            rc=ESMF_SUCCESS
*         endif
*
*        end subroutine ESMF_DeepValidate
```

```
        function ESMF_DeepCreate()
              type(ESMF_Deep) :: ESMF_DeepCreate

              ... other create code ...

*              ESMF_INIT_SET_CREATED(ESMF_DeepCreate)

        end function ESMF_DeepCreate

        subroutine ESMF_DeepDestroy(d)
              type(ESMF_Deep) :: d

              ... other create code ...

*              ESMF_INIT_SET_DELETED(d)

        end subroutine ESMF_DeepDestroy

        ... other routines ...

   end module ESMF_ExampleMod
```

7.10.6 Parameter Class

When adding a new parameter class definition, don't add any initialization standardization code. However, do add the new class to the class list in section 7.10.8.

7.10.7 Subroutine

When adding a new subroutine or function (both referred to as 'routine' henceforth) perform the following steps:

1. At the beginning of the routine, for each shallow class parameter S1 of type ESMF_STYPE. Add the line ESMF_INIT_CHECK_SHALLOW(ESMF_STYPEGetInit, ESMF_STYPEInit, S1). In addition, if the intent of S1 is intent(in), switch it to intent(inout) to allow it to be modified.

2. At the beginning of each routine, for each deep class *input* parameter D1 of type ESMF_DTYPE, add the line ESMF_INIT_CHECK_DEEP(ESMF_DTYPEGetInit, D1, rc). Here rc is the return code variable for the routine.

The initialization macros in 1. and 2. should be added before any code which uses the types being checked.

When adding the check macros to code there are a couple of issues to keep in mind for compatibility with all compilers. First, don't break up the macro across lines (e.g. using &). Second, some compilers have a maximum line length. Occasionally, the Deep class check macro will expand to larger than this length, if you find that this is occurring with a particular line use the ESMF_INIT_CHECK_DEEP_SHORT macro instead. It takes exactly the same parameter list as the normal deep class check macro, but expands to a much shorter line.

Here is an example illustrating this procedure:

Starting with this routine:

```
 subroutine ESMF_EXAMPLE(s1,d1,s2,d2,d3,rc)
       type(ESMF_Shallow1), intent(in) :: s1
       type(ESMF_Shallow2), intent(out) :: s2
```

```
          type(ESMF_Deep1),intent(in) :: d1
          type(ESMF_Deep2),intent(inout) :: d2
          type(ESMF_Deep3),intent(out) :: d3
          integer :: rc
          .... local variable declarations ...

          ! initialize return code
          rc=ESMF_FAILURE

           ...... rest of subroutine code....

  end subroutine ESMF_Example
```

The standardization yields this:
(modified lines marked with *)

```
   subroutine ESMF_EXAMPLE(s1,d1,s2,d2,d3,rc)
*          type(ESMF_Shallow1), intent(inout) :: s1
          type(ESMF_Shallow2) :: s2
          type(ESMF_Deep1) :: d1
          type(ESMF_Deep2),intent(inout) :: d2
          type(ESMF_Deep3),intent(out) :: d3
          integer :: rc
          .... other local variable declarations ...

          ! initialize return code
          rc=ESMF_FAILURE

*          ! check variables
*          ESMF_INIT_CHECK_DEEP(ESMF_Deep1GetInit,d1,rc)
*          ESMF_INIT_CHECK_DEEP(ESMF_Deep2GetInit,d2,rc)
*
*          ESMF_INIT_CHECK_SET_SHALLOW(ESMF_Shallow1GetInit,ESMF_Shallow1Init,s1)
*          ESMF_INIT_CHECK_SET_SHALLOW(ESMF_Shallow2GetInit,ESMF_Shallow2Init,s2)

           ...... rest of subroutine code....

  end subroutine ESMF_Example
```

7.10.8 ESMF Class Types

CAUTION: The following lists are very much outdated!!!

7.11 Code: Data Type Consistency Guidelines

The following tools and guidelines are in place in order to maintain the ESMF framework's portability and robustness, in light of the fact that neither the Fortran nor C++ standards require a fixed size for any given data type and/or kind. Please note that these guidelines pertain to the internal ESMF code, not to the user code that interfaces to it.

7.11.1 Use ESMF names for data types in C and ESMF data kinds in Fortran

7.11.2 Fortran

Any occurrence of Fortran data kind parameters within ESMF code must be specified with ESMF parameters only. e.g.

```
ESMF_KIND_I4
ESMF_KIND_I8
ESMF_KIND_R4
ESMF_KIND_R8
```

These are internally defined such that the size of the data they describe is the number at the end of the parameter. e.g. in

```
integer(ESMF_KIND_I8) :: someInteger
```

someInteger is an 8 byte integer in any platform.

Kind of literal constants:

Only the ESMF kind parameters listed above are allowed when specifying the kind of literal Fortran constants. e.g.

```
real(ESMF_KIND_R8) :: a,b
a=b*0.1_ESMF_KIND_R8
```

In the example above `0.1_ESMF_KIND_R8` is a literal constant of kind `ESMF_KIND_R8` and value 0.1.

This means that any other syntax, such as:

```
a=b*0.1d0
a=b*0.1_8
```

is NOT to be used.

`ESMF_KIND_I`, `ESMF_KIND_R`: Note that the `ESMF_TypeKind_Flag` parameters: `ESMF_KIND_I` and `ESMF_KIND_R` are defined for use in user code only. They are handles offered to the user for ESMF internal default kinds and not to be used internally in the ESMF framework.

Real Data:

1. All real data in Fortran should be declared with an explicit ESMF defined kind parameter in the form:

   ```
   real(ESMF_KIND_R<n>) :: RealVariable
   ```

   where $< n >$ is the size in bytes of the memory occupied by each real datum. Allowed values of n are those that correspond to supported type/kind data, e.g.

   ```
   real (ESMF_KIND_R8) :: a,b
   ```

   declares a and b to be 8 byte reals

2. Routines should be overloaded for all supported kinds(sizes) of real user data (e.g. in Array class routines).

3. Real parameter arguments, such as the arguments to grid or regrid routines need not be cause for overloading. They should be declared with the kind required for appropriate accuracy and uniform behavior across platforms. Documentation should identify when specific data types and kinds are required in calling such routines.

Integer Data:

The standards for integer data differ between integers that store user data and integers that do not.

Routines with user integer data arguments (e.g. integer arrays), must be overloaded for all integer data kinds supported. These will be declared as

```
integer(ESMF_KIND_I<n>)
```

where $< n >$ denotes data size in byte units.

Integers that do not represent user data must generally be declared without a specific kind (see the section below on Default Type Representation). Possible exceptions to this rule follow on items (3) and (4) below.

Care should be exercised to insure that integer arguments to intrinsic Fortran functions are of the correct kind (usually default). Likewise, argument parameters to VM routines should be of default kind.

In the case where a specific kind is required for integer ESMF routine arguments, such as occurs in TimeMgr routines, the requirement must be clearly documented (as noted above, ESMF kind parameters are to be used to specify the kind).

Default Type Representation:

Default Integer Data Internal to the ESMF:

Internally in the ESMF framework, all integer data intended to be of default kind must be declared without specifying their kind, e.g.

```
subroutine ESMF_SomeRoutine(foo)
integer :: foo
```

This is for clarity purposes, as well as to minimize the overall size of the framework.

`ESMF_KIND_I` must not be used to declare data internally within ESMF.

To provide some background, the handle `ESMF_KIND_I` is provided to aid users who want to insure that their integer user code matches the kind of ESMF internal default integer data. It is particularly useful when user code is compiled with integer autopromotion. User code that calls ESMF routines with internal default integer arguments could declare those integers to be of `kind=ESMF_KIND_I`, e.g.

```
program myUserCode
integer(ESMF_KIND_I) :: my_foo
call ESMF_SomeRoutine(my_foo)
```

Logical Data:

The standards for logical data differ between logicals which are used as dummy arguments in user-callable entry points, and those which need to be passed to ESMF C++ routines.

Routines with user logical data arguments use default kind arguments:

```
logical :: isActive
```

Local logicals within Fortran code also generally use the default Fortran logical data type.

ESMF defines a `ESMF_Logical` derived type for use when passing logical values between Fortran and C++ code. The constants `ESMF_TRUE` and `ESMF_FALSE` are available, and must be used to ensure compatibility with the C++ code.

For convenience, defined assignment operators are available which convert between Fortran logical and the `ESMF_Logical` derived type. Finally, there are .eq. and .ne. operators defined to compare `ESMF_Logical` values.

```
subroutine ESMF_SomeRoutine(myFlag)
  use ESMF
  implicit none

  logical, intent(inout) :: myFlag
```

```
  type(ESMF_Logical) :: myFlag_local

  myFlag_local = myFlag                   ! Defined assignment
  if (myFlag_local .eq. ESMF_TRUE) then  ! Use of .eq. operator
    call c_ESMC_SomeCRoutine (myFlag_local)  ! Pass to C++
  else
    call c_ESMC_OtherCRtoutine (myFlag_local)
  end if
  myFlag = myFlag_local                   ! Defined assignment
end subroutine
```

7.11.3 C and C++

ESMF C datatypes are also declared with the property that their size remains constant across platforms. This convention is set in order to make ESMF more robust in terms of portability, as well as for the correct matching of Fortran-C routine interfaces. They have a direct size correspondence to their Fortran counterparts. Their names are:

```
ESMC_I4 (=) integer type of size 4 bytes
ESMC_I8 (=) integer type of size 8 bytes
ESMC_R4 (=) floating point type of size 4 bytes
ESMC_R8 (=) floating point type of size 8 bytes
```

e.g.

```
ESMC_R4 someFloat;
```

here `someFloat` is a 4 byte floating point in any platform.

Real Data:

1. All real data in C should be declared with an explicit ESMF type in the form:

   ```
   ESMC_R<n>  RealVariable;
   ```

   where $< n >$ is the size in bytes of the memory occupied by each real datum. (Allowed values of n are those that correspond to supported type/kind data). e.g.

   ```
   ESMC_R8  a,b;
   ```

   which declares a and b to be 8 byte reals.

2. Methods containing user data - such as arrays - as arguments must handle all supported real user data types and sizes.

Integer Data:

1. All integers that do not represent user data must be declared using the C datatype `int`.
2. Methods containing user data - such as arrays - as arguments must handle all supported integer user data types and sizes.

Boolean Data:

1. Boolean values used internally within the C++ routines may use the `bool` data type.
2. Methods which are Fortran-callable, and pass boolean values through the dummy argument list, must use the `ESMC_Logical` enumerated type. The constants `ESMF_TRUE` and `ESMF_FALSE` are available for comparison purposes. These constants must be used to ensure compatibility with the Fortran code.

```
void fortran_caller(bool &myFlag) {
  ESMF_Logical myFlag_local;

  myFlag_local = myFlag;
  if (myFlag_local == ESMF_TRUE)
    ESMF_SomeFortranRoutine (&myFlag_local);  // Pass to Fortran
  else
    ESMF_OtherFortranRtoutine (&myFlag_local);

  myFlag = myFlag_local;
}
```

7.11.4 ESMF_TypeKind_Flag "labels": When knowing the data type/kind is necessary

An enumerated parameter in C++ and a corresponding sequence parameter in Fortran are provided in order to identify the type-and-kind of data being processed in various ESMF routines. There is a one to one correspondence between the Fortran sequence and the C++ enumerated type. They are mostly used as a tool to customize code for a specific data type and kind, and to insure the correct handling of user data across Fortran-C interfaces.

In Fortran this sequenced type's name is `ESMF_TypeKind_Flag`.

The `ESMF_TypeKind_Flag` parameter names declared are:

```
ESMF_TYPEKIND_I4   (4 byte integer)
ESMF_TYPEKIND_I8   (8 byte integer)
ESMF_TYPEKIND_R4   (4 byte floating point)
ESMF_TYPEKIND_R8   (8 byte floating point)
```

The following are supported for user interface use only (they are not to be used internally in the ESMF framework):

```
ESMF_TYPEKIND_I    (labels integer_not_autopromoted data)
ESMF_TYPEKIND_R    (labels real_not_autopromoted data)
```

Fortran `ESMF_TypeKind_Flag` parameters mostly appear in calls to C routines that create or manipulate arrays, fields, or bundles. e.g.

```
....
ESMF_TypeKind_Flag :: tk
...
 !-set tk
...
call FTN_X(c_ESMC_VMAllFullReduce( ..., tk, ...  )
...
...
```

Likewise in C the enumerated type's name is `ESMC_TypeKind_Flag`.

The `ESMC_TypeKind_Flag` parameter names declared are:

```
ESMC_TYPEKIND_I4   (4 byte integer)
ESMC_TYPEKIND_I8   (8 byte integer)
ESMC_TYPEKIND_R4   (4 byte floating point)
ESMC_TYPEKIND_R8   (8 byte floating point)
```

In C, `ESMC_TypeKind_Flag` parameters are used to tailor computation to the type and kind of user data arguments being processed. e.g

```
void FTN_X(c_esmc_vmallfullreduce)(.., ESMC_TypeKind_Flag *dtk, ..,int *rc){
  .....
  switch (*dtk){
  case ESMC_TYPEKIND_I4:
    ....
    break;
  case ESMC_TYPEKIND_R4:
    ....
    break;
  case ESMC_TYPEKIND_R8:
    ...
    break;
  default:
    ....
  }
  ...
```

7.11.5 Guidelines for Fortran-C Interfaces

There must be a one to one correspondence of Fortran and C arguments as illustrated in the table below. So for example, if a Fortran routine calls a C method that expects an argument of type `ESMC_I4`, the argument in the Fortran call should be declared to be an integer of `kind=ESMF_KIND_I4`.

```
integer(ESMF_KIND_I4)  <-->  ESMC_I4  [4 bytes]
integer(ESMF_KIND_I8)  <-->  ESMC_I8  [8 bytes]
real(ESMF_KIND_R4)     <-->  ESMC_R4  [4 bytes]
real(ESMF_KIND_R8)     <-->  ESMC_R8  [8 bytes]
integer                <-->  int
type(ESMF_Logical)     <-->  ESMC_Logical
etc.
```

Also, `ESMF_TypeKind_Flag` and `ESMC_TypeKind_Flag` values are set to have matching values. That is:

```
ESMF_TYPEKIND_I<n> = ESMC_TYPEKIND_I<n>
ESMF_TYPEKIND_R<n> = ESMC_TYPEKIND_R<n>
```

Character strings:

When passing character strings to subprograms, most Fortran compilers pass 'hidden' string length arguments by value after all of the user-supplied arguments. Each hidden argument corresponds to each character string that is passed. Because of varying compiler support of 32-bit vs 64-bit string lengths, the `ESMCI_FortranStrLenArg` macro is used in C++ code to specify the data type of the hidden arguments.

For input arguments is usually convenient to copy the Fortran string into a C++ string. The `ESMC_F90lentrim` procedure provides a common way to obtain the length of the Fortran string - while ignoring trailing blanks. It is analogous to the Fortran `len_trim` intrinsic function.

```
void FTN_X(c_esmc_somecode) (
    const char *name1,   // in - some name in a character string
    const char *name2,   // in - some other name
    int *rc,             // out - return code
    ESMCI_FortranStrLenArg name1_l,   // in - hidden length of name1
    ESMCI_FortranStrLenArg name2_l) { // in - hidden length of name2

    string name1_local = string (name1, ESMC_F90lentrim (name1, name1_l));
```

```
    string name2_local = string (name2, ESMC_F90lentrim (name2, name2_l));
    ...
}
```

Likewise, when C++ code needs to pass a character string to Fortran code, passing the string itself is performed by it address. The hidden length argument uses the standard string `length` method:

```
    FTN_X(f_esmf_somef90code) (
        name1_local.c_str(),
        name2_local.c_str(),
        &localrc,
        (ESMCI_FortranStrLenArg) name1_local.length(),
        (ESMCI_FortranStrLenArg) name2_local.length())
```

Note that when an array of characters is passed, the hidden string size is that of an individual array element. The size of each dimension of the array should be explicitly passed through separate arguments. Thus, in the following declaration, the hidden argument would have a value of 32. The dimensions, 20 and 30, would need to be separately passed:

```
character(32) :: char_array(20,30)
:
call c_esmc_something (  &
    char_array, size (char_array,1), size (char_array, 2), &
    localrc)
```

7.12 Code: Optional Argument Conventions for the C/C++ API

7.12.1 Overview

Optional arguments in the public ESMC interface are implemented using the variable-length argument list functionality provided by `<stdarg.h>`. The `<stdarg.h>` header contains a set of macros which allows portable functions that accept variable-length argument lists (also known as variadic functions) to be written. Variadic functions are declared with an ellipsis in place of the last parameter and must have at least one named (fixed) parameter. A typical example of such a function is printf which is declared as

```
int printf(char *fmt, ...).
```

The following type and macros are provided by `<stdarg.h>` for processing variable-length argument lists.

- `va_list`
  Type suitable for use in accessing the variable-length argument list of a function with the stdarg macros;

- `void va_start(va_list ap, last_arg)`
  Initializes the variable argument list pointer *ap* to the beginning of the variable argument list, before any calls to `va_arg`. *last_arg* is the last fixed argument being passed to the function (the argument before the ellipsis).

- *type* `va_arg(va_list ap,` *type*`)`
  Returns the next argument in the variable-length argument list pointed to by `ap`. Each invocation of `va_arg` modifies `ap` such that the values of successive arguments are returned in turn. The *type* parameter is the type the argument is expected to be. The parameter *type* is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a `*` to *type*. Different types can be mixed, but it is up to the function to know what type of argument is expected. Note that `ap` must be initialized with `va_start`. If there is no next argument, then the result is undefined.

- `void va_end(va_list ap)`
  Destroys the variable argument list pointer `ap`, rendering it invalid unless `va_start` is called again. Each invocation of `va_start` must be matched by a corresponding invocation of `va_end` in the same function. After the call `va_end(ap)` the variable `ap` is undefined. Multiple traversals of the list, each bracketed by `va_start` and `va_end` are possible.

Arguments corresponding to the variable-length argument list specified by `",..."` in a function prototype always undergo the following argument conversions: `bool` or `char` or `short` are converted to `int`, and `float` is converted to `double`.

In order to correctly retrieve arguments from the variable argument list it is necessary to know the type of each argument. In order to correctly terminate parsing the variable argument list it is also necessary to know the number of arguments in the list. Since the number of arguments and their types are not known at compile time the best one can do is establish a convention such that the caller will provide type and length information about the variable argument list. The convention, however, is not a guarantee that the caller does not mistakenly specify the types or number of arguments. The `scanf` and `printf` functions do this by having the last fixed argument be a character (format) string in which the caller specifies the types (for example, `"%f%d%s"` for each argument in the variable argument list. Although, the format string specifies the number of expected arguments there is no actual "termination-flag" inserted at the end of the variable argument list. If the caller incorrectly specifies the in the format string more arguments than are actually provided then the function will likely access invalid sections of memory while parsing the variable argument list. Unless this error is trapped by the system the function will unwittingly process erroneous data.

7.12.2 Approach

The following conventions are established for the public ESMC optional argument API. The "class" associated with this API is called `ESMC_Arg` and its header files are located in `Infrastructure/Util/include`. The public optional argument API for a class is provided by the public header for that class together with `ESMC_Arg.h`. Macros for internal processing of optional arguments are provided by the `ESMCI_Arg.h` header. Each ESMC class has a set of named optional arguments associated with its methods. Associated with each optional argument of a class is a unique (within the class) optional argument identifier (*id*) of type `ESMCI_ArgID`. The optional argument list provided by the caller must consist of a sequence of argument identifier and argument (*id*, *arg*) pairs. The optional argument list must be terminated by the global identifier `ESMCI_ArgLastID`. The `ESMC_ArgLast` macro is provided by `ESMC_Arg.h` for the user to indicate the end of an optional argument list.

The global identifier `ESMCI_ArgBaseID` is the starting identifier for local (class) optional argument identifiers. `ESMCI_ArgBaseID` establishes a set of global optional argument identifiers, such that, the global set does not intersect with any class identifier set. The optional argument identifier list for a class will be declared in the public header for that class. The naming convention for optional argument identifiers is

```
ESMCI_<ClassName>Arg<ArgName>ID
```

where `ClassName` is the name of the class and `ArgName` is the name of the optional argument with the first letter capitalized. The optional argument identifier list is declared as enumeration constants with the first identifier set equal to `ESMCI_ArgBaseID`. Here is an example for a class called "Xclass".

```
// Optional argument identifier list for the ESMC_Xclass API.
enum {
  ESMCI_XclassArgAoptID = ESMCI_ArgBaseID,  // ESMC_I1
  ESMCI_XclassArgBoptID,                    // ESMC_I4
  ESMCI_XclassArgCoptID,                    // ESMC_R4
  ESMCI_XclassArgDoptID,                    // ESMC_R8
  ESMCI_XclassArgEoptID,                    // ESMC_Fred
};
```

It is helpful for the class developer to list the data type of each optional argument (as a comment) with its associated *id*.

The `ESMCI_Arg(ID,AR)` internal macro (declared in `ESMC_Arg.h`) is provided for casting an optional argument and its associated *id* into the required sequence for passing to functions.

```
#define ESMCI_Arg(ID,ARG)   ((ESMCI_ArgID)ID),(ARG)
```

Each class will use this internal macro in its public header to declare user macros for the class specific optional arguments. The naming convention for class specific optional argument expansion macros is

```
ESMC_<ClassName>Arg<ArgName>(ARG)
```

where `ClassName` is the name of the class and `ArgName` is the name of the optional argument with the first letter capitalized. The argument of the macro is the actual caller provided optional argument. Here is an example of how the macros are used for the class "Xclass".

```
// Argument expansion macros for the ESMC_Xclass API.
#define ESMC_XclassArgAopt(ARG)  ESMCI_Arg(ESMCI_XclassArgAoptID,ARG)
#define ESMC_XclassArgBopt(ARG)  ESMCI_Arg(ESMCI_XclassArgBoptID,ARG)
#define ESMC_XclassArgCopt(ARG)  ESMCI_Arg(ESMCI_XclassArgCoptID,ARG)
#define ESMC_XclassArgDopt(ARG)  ESMCI_Arg(ESMCI_XclassArgDoptID,ARG)
#define ESMC_XclassArgEopt(ARG)  ESMCI_Arg(ESMCI_XclassArgEoptID,ARG)
```

Here is an example call, with properly specified optional arguments, to a class function. The function has three named (fixed) arguments.

```
rc = ESMC_XclassFunc(fixedArg1,fixedArg2,fixedArg3,
                     ESMC_XclassArgAopt(aOptArg),
                     ESMC_XclassArgCopt(cOptArg),
                     ESMC_XclassArgDopt(dOptArg),
                     ESMC_ArgLast)
```

7.12.3 Internal Macros for Processing the Optional Argument List

Internal macros for processing the optional argument list are declared in `ESMCI_Arg.h`. These macros provide a consistent internal interface to the macros defined in `stdarg.h`. What follows is a description of each macro available to a class developer.

- `ESMCI_ArgList`
  Optional argument list data type (`va_list`). The variable used for accessing the optional argument list must be declared as this type.
- `ESMCI_ArgStart(AP,LAST)`
  Internal macro to initialize optional argument list processing. `AP` is the optional argument list pointer. `LAST` is the last fixed argument (before the optional argument list).
- `ESMCI_ArgEnd(AP)`
  Internal macro to finalize optional argument list processing. `AP` is the optional argument list pointer.
- `ESMCI_ArgGetID(AP)`
  Internal macro to return an optional argument identifier. `AP` is the optional argument list pointer.

The following type specific `ESMCI_ArgGet` macros are provided for returning optional argument values. These macros correctly account for the forced type conversions described above. In each macro, `AP` is the optional argument list pointer.

Internal macros for returning standard C (non-pointer) types.

```
#define ESMCI_ArgGetChar(AP)           (char)va_arg(AP,int)
#define ESMCI_ArgGetShort(AP)         (short)va_arg(AP,int)
#define ESMCI_ArgGetInt(AP)             (int)va_arg(AP,int)
#define ESMCI_ArgGetLong(AP)           (long)va_arg(AP,long)
#define ESMCI_ArgGetLongLong(AP)  (long long)va_arg(AP,long long)
#define ESMCI_ArgGetFloat(AP)         (float)va_arg(AP,double)
#define ESMCI_ArgGetDouble(AP)       (double)va_arg(AP,double)
```

Internal macros for returning defined ESMC types.

```
#define ESMCI_ArgGetI1(AP)          (ESMC_I1)va_arg(AP,int)
#define ESMCI_ArgGetI2(AP)          (ESMC_I2)va_arg(AP,int)
#define ESMCI_ArgGetI4(AP)          (ESMC_I4)va_arg(AP,int)
#define ESMCI_ArgGetI8(AP)          (ESMC_I8)va_arg(AP,ESMC_I8)
#define ESMCI_ArgGetR8(AP)          (ESMC_R8)va_arg(AP,double)
#define ESMCI_ArgGetR4(AP)          (ESMC_R4)va_arg(AP,double)
```

Special internal macro for returning strings.

```
#define ESMCI_ArgGetString(AP)              va_arg(AP,char*)
```

General internal macro for returning all non-converted types.

```
#define ESMCI_ArgGet(AP,TYPE)               va_arg(AP,TYPE)
```

`TYPE` is the expected type of returned argument.

7.12.4 Parsing the Optional Argument List

Each class function with optional arguments will declare an optional argument list pointer and optional argument list *id*.

```
ESMCI_ArgList argPtr;       // optional argument list pointer
ESMCI_ArgID argID;          // optional argument list id
```

Parsing the optional argument list consists of three distinct phases: initializing the argument list pointer, retrieving each argument in succession based on its *id*, and finalizing the argument list pointer. The block for retrieving arguments must first retrieve an argument *id*. If that *id* is not `ESMCI_ArgLastID` and is in the set of valid ids for that function, then the argument is retrieved using a type-appropriate `ESMCI_ArgGet` macro. The following example code should be considered as "template" code for parsing an optional argument list.

```
// parse the optional argument list:
ESMCI_ArgStart(argPtr,ifix);
while ( (argID = ESMCI_ArgGetID(argPtr)) != ESMCI_ArgLastID ) {
  switch ( argID ) {
    case ESMCI_XclassArgAoptID:
      aOpt = ESMCI_ArgGetI1(argPtr);
      break;
    case ESMCI_XclassArgBoptID:
      bOpt = ESMCI_ArgGetI4(argPtr);
      break;
    case ESMCI_XclassArgCoptID:
      cOpt = ESMCI_ArgGetR4(argPtr);
      break;
    case ESMCI_XclassArgDoptID:
```

```
      dOpt = ESMCI_ArgGetR8(argPtr);
      break;
    case ESMCI_XclassArgEoptID:
      eOpt = ESMCI_ArgGet(argPtr,ESMC_Fred);
      break;
    default:
      ESMC_LogDefault.MsgFoundError(ESMC_RC_OPTARG_BAD, "", &rc);
      return rc;
  } // end switch (argID)
} // end while (argID)
ESMCI_ArgEnd(argPtr);
```

Prior to actually parsing the optional argument list a function must check the list for proper specification. This is done by using a code block similar to the parsing block shown above. The difference is that in this case the `ESMCI_ArgGet` macros are only used to increment the argument list pointer.

```
// check the optional argument list:
ESMCI_ArgStart(argPtr,lastFixed);
while ( (argID = ESMCI_ArgGetID(argPtr)) != ESMCI_ArgLastID ) {
  switch ( argID ) {
    case ESMCI_XclassArgAoptID: ESMCI_ArgGetI1(argPtr); break;
    case ESMCI_XclassArgBoptID: ESMCI_ArgGetI4(argPtr); break;
    case ESMCI_XclassArgCoptID: ESMCI_ArgGetR4(argPtr); break;
    case ESMCI_XclassArgDoptID: ESMCI_ArgGetR8(argPtr); break;
    case ESMCI_XclassArgEoptID: ESMCI_ArgGet(argPtr,ESMC_Fred); break;
    default:
      ESMC_LogDefault.MsgFoundError(ESMC_RC_OPTARG_BAD, "", &rc);
      return rc;
  } // end switch (argID)
} // end while (argID)
ESMCI_ArgEnd(argPtr);
```

The following example shows how an optional argument list check block can be structured to handle an optional argument whose type is call dependent. In this case the type of the `dValue` optional argument depends on the value of the `ESMC_TypeKind_Flag` argument `tk` (which is the last fixed argument).

```
// check the optional argument list:
ESMCI_ArgStart(argPtr,tk);
while ( (argID = ESMCI_ArgGetID(argPtr)) != ESMCI_ArgLastID ) {
  switch ( argID ) {
    case ESMCI_ConfigArgCountID: ESMCI_ArgGetInt(argPtr); break;
    case ESMCI_ConfigArgLabelID: ESMCI_ArgGetString(argPtr); break;
    case ESMCI_ConfigArgDvalueID:
      switch ( tk ) {
        case ESMC_TYPEKIND_I4: ESMCI_ArgGetI4(argPtr); break;
        case ESMC_TYPEKIND_I8: ESMCI_ArgGetI8(argPtr); break;
        case ESMC_TYPEKIND_R4: ESMCI_ArgGetR4(argPtr); break;
        case ESMC_TYPEKIND_R8: ESMCI_ArgGetR8(argPtr); break;
        case ESMC_TYPEKIND_LOGICAL: ESMCI_ArgGetInt(argPtr); break;
        case ESMC_TYPEKIND_CHARACTER: ESMCI_ArgGetString(argPtr); break;
      } // end switch (tk)
      break;
    default:
```

```
      ESMC_LogDefault.MsgFoundError(ESMC_RC_OPTARG_BAD, "", &rc);
      return rc;
    } // end switch (argID)
  } // end while (argID)
  ESMCI_ArgEnd(argPtr);
```

7.13 Code: Makefile Conventions

The makefiles use GNU standard target names when possible. The default rule is to remake the ESMF library. Targets exist to build the unit and system tests, the examples, the demos, and to run them together or individually. Targets also exist to build the documentation and create pdf and html versions. For more details on targets, refer to the README file in the top level ESMF directory, and also the *ESMF User's Guide*.

7.13.1 Code Building Rules

During software development it is advantageous to recompile a portion of ESMF and not build the entire framework when testing localized software modifications. The makefiles are configured to compile only files in the current directory and rebuild the shared libraries only when necessary. Therefore when "gmake" is entered in any directory, only files in that directory are compiled. However, the entire ESMF framework may be built from any directory by entering "gmake lib".

The unit tests, the system tests, the examples, and the demos all share a common set of targets. The following target list illustrates the options for the examples. The names `unit_tests`, `system_tests`, `demos`, and `all_tests` can be substituted wherever `examples` occurs.

examples build and run all examples

examples_uni build and run all examples single process

build_examples build the examples

run_examples run the examples and report the success/fail status

run_examples_uni run the examples single process

check_examples report the success/fail status without reexecution

clean_examples remove the example executables

7.13.2 Document Building Rules

The makefile rules for building documents are now located in:

```
build/common.mk
```

The rules exist as pattern rules and are controlled by the variables set in a doc directory's makefile. The pattern rules will build .tex from source code and .pdf, .dvi and html files from .ctex files.

Variables that can be set in the individual doc/makefiles are called "makefile variables". These are:

DVIFILES

PDFFILES

HTMLFILES

TEXTFILES_TO_MAKE

Document file dependencies are set with:

REFDOC_DEP_FILES

The makefiles targets work from both local directory and from ESMF_DIR:

gmake alldoc

gmake pdf

gmake dvi

gmake html

7.13.3 Include Files

At the highest directory level, the "include" directory will contain public include files intended to be used in user-written code. They will be broken up into separate files, with a single "ESMF.h" include file which includes all the others.

At the 'src' level, parallel to Infrastructure, Superstructure, and doc, is another include directory. This is for private include files which are ESMF-wide. These might include system-wide architecture dependent #defines constants, etc.

Below the Superstructure & Infrastructure level are the individual component levels (TimeMgr, Field, Component, etc). Under each of these directories is an include directory for the component specific include files.

7.14 Preprocessor Usage

7.14.1 Using the Preprocessor For Generic Fortran Code

Fortran allows the creation of generic subprogram names via the `INTERFACE` facility. This lets the caller use a simplified single name for a call. Then, based on the callers actual argument list, the compiler will choose an appropriate specific routine to perform the task.

When several specific variants of a generic call exist, and only differ in minor ways, for example by ESMF object type, it can be desirable to have the variants generated from a single definition. This allows improvements in the reliability and maintainability of the code. Feature code can be added in a single place, and all of the specific routines will automatically incorporate the new code.

By convention, generic ESMF Fortran code uses the GNU C preprocessor macro facility to generate specific routines. These routines generally differ only in the type, kind, and rank of the dummy arguments. The internal code is otherwise identical.

When writing a cpp macro, the definition must be 'one line' long. Since coding an entire subprogram requires many lines to implement, a backslash character is used at the end of each line to concatenate multiple lines from the input file into a single, very long, preprocessor line. After macro expansion, each preprocessor line needs to be converted into a sequence of Fortran statements. So again by convention, at-sign characters, @, are used as the second-to-last character in each line. After cpp has been run, the long lines from the result are split at the @ characters by converting them to newline characters. The `tr` command is used for this translation.

ESMF Fortran files which need to be preprocessed in this manner use the file name suffix `.cppF90`. This triggers the makefile system to use the following sequence when processing the file:

- The gcc version of cpp is run. This causes a first pass of preprocessing to occur. All macro expansion for generic routines should be performed in this pass.
- The `tr` command is run to perform several transformations:
    1. Convert @ characters to newline characters. As described above, this converts the one very long line of any cpp macro expansion into multiple source lines for the compiler.
    2. Convert caret, ^ characters to # characters in order to allow selected preprocessing directives to pass through to the second level of preprocessing performed by the actual compiler.

3. Convert vertical bar, |, characters to apostrophe (single quote) characters. Vertical bars should be used where apostrophe characters need to appear. This is needed because some versions of cpp look for matching apostrophes in order to properly issue code. Note that only lines within a macro definition, i.e., those ending in @ need to use this transformation.

- The Fortran compiler is run. The built-in preprocessor is used for the second preprocessing pass. This pass is used for other preprocessing tasks - such as including header files needed for compilation, or for conditional compilation of system-dependent code.

A simple example of generic code written in this style is:

```
  module ESMF_Example_mod
    use ESMF
    implicit none

! The following header needs to be preprocessed by the compiler,
! not the first cpp pass.  So use a caret as the first character,
! rather than a pound-sign.

^include "my_header.inc"

! Define a generic name with three specific routines

    interface ESMF_generic_name
      module procedure ESMF_ArraySpecifc
      module procedure ESMF_FieldSpecfic
      module procedure ESMF_MeshSpecific
    end interface

  contains

! Define a macro for expansion.  Each line is terminated with a
! @\ sequence.  Also note the use of ## for concatenation
! and # for stringization.

#define MySpecificMacro(mclass, mthis)  \
  subroutine mclass##Specific (mthis, rc) @\
    type(mclass), intent(inout) :: mthis @\
    integer, optional :: rc @\
 @\
    print *, |class = |, #mclass @\
 @\
    if (present (rc)) rc = ESMF_SUCCESSFUL @\
 @\
  end subroutine mclass##Specfic @\

! Expand macro for a few classes

MySpecificMacro(ESMF_Array, array)
MySpefificMacro(ESMF_Field, field)
MySpecificMacro(ESMF_Mesh, mesh)
```

```
end module ESMF_Example_mod
```

Character string concatenation via the Fortran concatenation operator // can be problematic due to the first pass preprocessing treating it as a C++ inline comment. An alternative is to use the ESMF_StringConcat function.

7.14.2 System Dependent Strategy Using Preprocessor

Since the code must compile across different platforms, a strategy must be adopted to handle the system differences. Examples of system differences are: the subroutines (bcopy vs memcopy) or include filenames (strings.h vs str.h, etc).

Rather than putting architecture names in all the source files, there will be an ESMF-wide file which contains sections like this:

```
#ifdef sgi
#define HAS_VFORK 0
#define BCOPY_FASTER 1
#define FOPEN_3RD_ARG 1
#endif

#ifdef sun
#define HAS_VFORK 1
#define BCOPY_FASTER 0
#define FOPEN_3RD_ARG 1
#endif
```

This allows system-dependent code to be bracketed with meaningful names:

```
#if HAS_VFORK
   vfork();
#else
   fork();
   exec();
#endif
```

and not an endless string of architecture names.

7.15 ESMF Data Type Autopromotion Support Policy and Guide

ESMF supports autopromotion of user code with respect to integer and real data types. This is data that a user would put in an array or field. By autopromotion we mean compiling user code with $-r8$, $-i8$ or similar options on those platforms where these options are available. If a code is compiled with autopromotion, user data will be processed correctly by ESMF. That is because user data arguments in ESMF calls already are overloaded for all TKR(type-kind-rank). It is important to note that ESMF itself must not be compiled with autopromotion flags.

The ESMF API distinguishes four different flavors of integer and real arguments:

- integer parameters
- integer user data

- real parameters
- real user data

The ESMF API handles integer user data by overloading for TKR. The ESMF API handles real user data by overloading for TKR. The ESMF API handles integer parameters by using the system's default integer type. The ESMF API handles real parameters by explicitly specifying a kind.

As a result of the ESMF API conventions the autopromotion of integer and real user data is automatically handled within ESMF.

Integer parameters that are arguments to ESMF calls must be protected against integer autopromotion by explicitly specifying their kind as `ESMF_KIND_I` in the user code.

The safest way to handle real parameters is for the user to specify their explicit kind as expected by the API. However, the user may choose to omit the kind specifier for reasons of convenience. This however will make their code vulnerable with respect to autopromotion and make the user code less portable, breaking it where the system default real kind does not match the ESMF API.

The requirements for safe interfacing of autopromoted user code with ESMF are as follows:

1. All integer parameter arguments (as opposed to user data) to ESMF routines must be of kind `ESMF_KIND_I`. Following is an example of an ESMF call that includes both a user data argument, and integer parameter arguments

```
   ...

integer, dimension(:), pointer :: intptr        !User data arg.
                                                !->No need for
                                                !kind specifier.

integer(ESMF_KIND_I), dimension(3) :: counts    ! Integer
                                                ! parameter

integer(ESMF_KIND_I), dimension(3) :: lbounds   ! Integer
                                                ! parameter

integer(ESMF_KIND_I), dimension(3) :: ubounds   ! Integer
                                                ! parameter

integer(ESMF_KIND_I) :: rc                      ! Integer
                                                ! parameter
 ...
call  ESMF_LocalArrayCreate(array, counts, intptr, &
        lboounds, ubounds, rc)
```

2. As explained in section I.3 above, all real parameters must be of the kind expected by the routine being called. e.g. In a call to `ESMF_ClockGet()`, the `runTimeStepCount` argument must be of kind `ESMF_KIND_R8`.

```
 ...
type(ESMF_Clock)  :: clock
real(ESMF_KIND_R8) :: runTimeStepCount
integer(ESMF_KIND_I) :: rc
 ...
call ESMF_ClockGet(clock, runTimeStepCount, rc)
```

3. When user data is autopromoted, the ESMF_TypeKind_Flag parameter argument corresponding to the autopromoted data must be adjusted appropriately. The ESMF_TypeKind_Flag parameter for an array or field that has been autopromoted can be obtained during execution by using the function ESMF_TypeKindFind(). ESMF_TypeKindFind() is an overloaded function that returns the ESMF_TypeKind_Flag parameter corresponding to the runtime type and kind of an input scalar.

   e.g. In the following code excerpt ESMF_TypeKindFind() is used to determine at runtime the correct ESMF_TypeKind_Flag parameter to use in the ESMF_ArraySpecSet() routine in preparation for creation of an ESMF Array that will store integer data that may or may not be autopromoted.

```
...
integer :: iScalar
...
type(ESMF_TypeKind_Flag) myTypeKind
myTypeKind= ESMF_TypeKindFind(iScalar)
call ESMF_ArraySpecSet(arrayspec, rank, myTypeKind, rc)
```

7.15.1 How We Arrived at This Autopromotion Support Policy

We considered the possibility of expanding autopromotion support to ESMF routine integer parameter arguments, in addition to those storing user data. Two options were considered as explained below. The reasons why we decided against such support, after discussion with the ESMF community, are as follows:

1. All user interfaces would need to be overloaded -this in addition to the overloading already present on some routines for user data.
2. Most likely, cpp macros for all user interfaces would be needed, which does not improve readability.
3. The additional necessary overloading would double (or more, depending on the option) the size of our post-processor code base.
4. Typecasting support would incur performance overhead, regardless of whether autopromotion is used or not.
5. There are routines, such as ESMF_LocalArrayCreate(), where all integer parameter arguments are optional. Overloading those routines violates the Fortran 95 standard and will not compile.

Rejected Options:

We considered and rejected two alternate options for autopromotion support. Here we illustrate with an example their impact on both the user code and the ESMF code. Note that this routine is overloaded for data arrays for all TKR supported and thus supports both real and integer autopromotion of user data.

1st option: support autopromotion of integers (options sizes 4 or 8 bytes) as long as all integer parameter arguments are of the same kind.

2nd option: support autopromotion of integers (options sizes 4 or 8 bytes). Mixed integer argument kinds.

For our example we use a call to ESMF_LocalArrayCreate() in each of the 2 option scenarios. In reading these example scenarios please keep in mind that ESMF code will not be compiled with autopromotion.

That is because only user code autopromotion is being considered. Thus the following statement could have different meaning if it is found within ESMF than in user code:

```
                        integer :: foo

In ESMF--------------------------------- ===>  foo is of default
                                              integer kind/size.

In User code compiled with autopromotion ==>  foo is an integer
```

```
                                                    of kind/size
                                                    determined by
                                                    autopromotion flag.
```

With 1st Option:

```
USER CODE:
---- -----
    ...
   integer, dimension(:), pointer :: intptr
   integer             , dimension(3) :: counts
   integer             , dimension(3) :: lbounds
   integer             , dimension(3) :: ubounds
   integer              :: rc
    ...
    ...
   call  ESMF_LocalArrayCreate(array, counts, intptr, &
         lboounds, ubounds, rc)
```

Changes in ESMF:

The routine would need to be further overloaded in order to insure that all integer parameter call arguments (`counts, lbounds, ubounds,` and `rc`), are typecast to default integer inside the method (note that ESMF kind can be of size 4 or 8). It would be something like this:

```
 subroutine ESMF_LocalArrayCreate(array, counts, intptr,.... &
                     lbounds, ubounds, rc)
 ......
  integer(<esmfKind>) , intent(in), optional :: counts, &
                        lbounds, ubounds
  integer(<esmfKind>), intent(out), optional :: rc
  integer :: counts_noAP, lbounds_noAP, ubounds_noAP, rc_noAP

  if (PRESENT(counts)) counts_noAP=counts
  if (PRESENT(lbounds)) lbounds_noAP=lbounds
  if (PRESENT(ubounds)) ubounds_noAP=ubounds
 ...
 ....
  if (PRESENT(rc)) rc=rc_nonAP
 return
```

With $< esmfKind >$ = `ESMF_KIND_I4` in one copy and `ESMF_KIND_I8` in the other. The number of copies of the routine will increase from 42 to 84.

*(Note that in this particular routine, overloading is problematic because all the integer parameters are optional)

With 2nd Option:

```
USER CODE:
---- -----
    ...
   integer, dimension(:), pointer :: intptr
   integer(ESMF_KIND_I4) , dimension(3) :: counts
   integer(ESMF_KIND_I8), dimension(3) :: lbounds
```

```
integer             , dimension(3) :: ubounds
integer              :: rc
 ...
 ...
call  ESMF_LocalArrayCreate(array, counts, intptr, &
                 lboounds, ubounds, rc)
```

Changes in ESMF:

The routine would need to be further overloaded to support call arguments `counts, lbounds, ubounds,` and `rc` each being of either 4-byte or 8-byte size. In order to provide for all possible combinations of data sizes for these 4 integer parameters, the number of overloads would increase from 42 to $[42 * (2 * *4)] = 672$, which explains why support of this option is not practical.

7.16 Scripts: Script Coding Standard

7.16.1 Content Rules

- **Prologues** Scripts included in the ESMF should be identified with a header that includes the name of the script, a description, an interface summary, the author and the origination date.

7.17 Lang: Interlanguage Coding Conventions

ESMF is written in a combination of C/C++ and Fortran. Techniques used in ESMF for interfacing C/C++ and Fortran codes are described in the *ESMF Implementation Report*[9], which is available via the **Users** tab on the ESMF website. These techniques, which address issues of memory allocation, passing objects across language boundaries, handling optional arguments, and so on, are general and have been applied to multiple projects.

We distinguish between these techniques and the conventions used by the ESMF project when interfacing C/C++ and Fortran. These conventions, which represent specific implementation choices, require additional input and explanation, and this section in the *Guide* is currently incomplete. The list below outlines the topics that we intend to address:

1. Logicals across language interfaces
2. Optional arguments across language interfaces
3. Layering Fortran on top of C/C++
4. Layering C/C++ on top of Fortran

7.17.1 Optional Arguments Across Language Interfaces

It is often necessary for C++ code to call Fortran code where optional arguments are used. By convention, most Fortran compilers use a C NULL for the argument address to indicate that the optional argument is missing.

The following Fortran subroutine has optional arguments. Note that Fortran 77 style adjustable size array dimensioning is used for array b:

```
subroutine f_ESMF_SomeProcedure (a, b, b_size, rc)
  implicit none
  real,    intent(in),  optional :: a
  integer, intent(in)            :: b_size
  real,    intent(out), optional :: b(b_size)
  integer, intent(out), optional :: rc

  if (present (a)) then
```

```
    ... = a
  end if
  ...
  if (present (b)) then
    b(i) = ...
  end if
  ...
  if (present (rc)) then
    rc = localrc
  end if
end subroutine f_ESMF_SomeProcedure
```

When calling the above from C++, NULL can be used to indicate missing arguments:

```
extern "C" {
    FTN_X(f_esmf_someprocedure)(float &a, float &b, int &b_size, int &rc);
}
...
// array b not present, so use NULL.
int b_size = 0;
FTN_X(f_esmf_someprocedure)(&a, NULL, &b_size, &rc);
...
// array b is present.  Pass size and address.
float *b = new float[20];
int b_size = 20;
FTN_X(f_esmf_someprocedure)(&a, b, &b_size, &rc);
```

7.18 Lang: Fortran Coding Standard

This standard is derived from the GFDL Flexible Modeling System Developers' Manual and the coding standard for the NCAR Commun developed by Jim Rosinski. Other documents containing coding conventions include the "Report on Column Physics Standards" (http://nsipp.gsfc.nasa.gov/infra/) and "European Standards For Writing and Documenting Exchangeable Fortran 90 Code" (http://nsipp.gsfc.nasa.gov/infra/eurorules.html).

The conventions assume the use of embedded documentation extractor ProTeX.

7.18.1 Content Rules

F95 Standard ESMF will adhere to the Fortran 95 language standard [6], to the extent that it is implemented.

1. All elements of the ANSI `f95` standard are permitted, with a few listed exceptions whose use is discouraged or prohibited. These are enumerated below.
2. Language extensions are severely restricted. They may be used in limited fashion, provided a pressing reason exists (e.g major performance enhancement using a particular proprietary software system), *and* an alternate formulation is provided for compiling environments that do not permit the extension.
3. The standard may change in the future, e.g to Fortran 2003, or any other, after review.

Preprocessing The use of preprocessing directives is intended for language extensions, and in some circumstances, it is used to generate module procedures under a generic interface for variables of different type, kind and rank (thus circumventing `f90`'s strict typing), while maintaining a single copy of the source.

The use of preprocessor directives in ESMF is permitted under the following conditions:

1. Where language extensions are used, `cpp` `#ifdef` statements must be used to shield lines from compilers that may not recognize them.
2. Use is restricted to the built-in preprocessor of the `f90` compiler (based on `cpp`), and cannot be based on external preprocessors such as `m4`. This condition may be relaxed on platforms where the builtin preprocessor proves to be inadequate.
3. Use is restricted to short code sections (a useful rule of thumb is that an `#ifdef` and the matching `#endif` should both be visible on a single 80×24 editor window).
4. Tokens must be uppercase.
5. Owing to restrictions in certain compilers, preprocessor variable names may not exceed 31 characters.

Source files Each source code file defines a single `program` or `f90 module`. The filename must be the same as the module name with the following extensions:

```
filename.f - fixed format, no preprocessing.
filename.F - free format, no preprocessing.

filename.f90 - fixed format, with preprocessing.
filename.F90 - free format, with preprocessing.
```

Module name The names of Fortran procedure interfaces will be preceded by `ESMF_`. Class names will normally be the first item in a procedure name, followed by the specific method, e.g., `ESMF_TimeMgrGetCurrDate`. Compilers produce object code for each source file, usually with a `.o` extension. During linking, it is required that each object file have a unique name; extremely generic names must be avoided. This convention is used to prevent name collisions.

Scope Each module in ESMF must have `private` scope by default. Each public interface therefore needs to be explicitly published.

Typing The use of implicit typing is forbidden. Every module must contain the line:

```
implicit none
```

in the module header, and every variable explicitly declared.

There are a few restrictions on the length of a character variable:

1. Character variables that are *arguments* to routines should be declared with `(len=*)`. It has been observed that compilers are inconsistent in their "padding" practices, and the standard is silent on the subject.
2. It is recommended that other character variables be declared with length a multiple of 4, or preferably 8. This is a *requirement* for variables that are components of derived types, since it has been observed that without these restriction, there are occasional word alignment fault errors generated.

Arguments The `intent` of arguments to subroutines and functions must be explictly specified.

Intrinsics The `f90` language provides a number of intrinsic functions for performing common operations. The use of the standard intrinsics is generally encouraged. Notes:

1. The generic form of the intrinsic (e.g `max()`) must be used rather than the specific one (e.g `dmax0()`). This permits flexibility to later changes of type.

2. Many of the intrinsic array operations have been found to be poorly optimized for performance (e.g `reshape()`, `matmul()`) since they have to be perfectly general. These must be used with care in code regions that are critical for performance.
3. Several older standard intrinsic names have been declared obsolescent, and the current names are preferred (e.g `modulo()` instead of `mod()`, `real()` instead of `float()`).

Constants and magic numbers Shared constants must never be hardcoded: instead mnemonically useful names are required. This applies to physical constants such as the universal gas constant, gravity, and so on, but also for flags used to select code options. In particular, this coding construct:

```
subroutine advection(flag)
integer, intent(in) :: flag
...
if( flag.EQ.1 )then
    call upwind_advection( ... )
else if( flag.EQ.2 )then
    call smolar_advection( ... )
...
endif
end subroutine advection
...
call advection(1)
```

is discouraged. This should instead be written as:

```
integer, parameter :: UPWIND=1, SMOLAR=2
...
subroutine advection(flag)
integer, intent(in) :: flag
...
if( flag.EQ.UPWIND )then
    call upwind_advection( ... )
else if( flag.EQ.SMOLAR )then
    call smolar_advection( ... )
...
endif
end subroutine advection
...
call advection(UPWIND)
```

Procedural interfaces Procedural interfaces are the public interfaces to subroutines and functions provided by a module.

1. Procedures that perform the same function on different datatypes (e.g of differing type, kind or rank) should have a single generic interface. When the generic public interface exists, all the module procedures that constitute it must be private.
2. Optional arguments, if any, should *follow* the required arguments, so that the procedure may be called without explicit argument keywords. Optional arguments in all new public interfaces *must* have a `keywordEnforcer` argument separating the required arguments from the optional arguments. This requires the caller to specify keywords for the arguments, allowing for upward compatibility as new optional arguments are added over time.

3. Argument lists should be as short as possible. If necessary, related elements of an argument list should be encapsulated in a public derived type.

Deprecated elements of the standard Deprecated language elements include:

1. implicit typing. Use `implicit none` in all modules and external procedures.
2. `common` blocks. Use module global variables instead;
3. assumed size arrays: i.e declarations of the form `a(*)` or `a(1)` with the intention of over-indexing. This can inhibit effective bounds-checking at compile- and runtime.
4. `STOP` statements: this can generate single-processor exits in some parallel environments;
5. array syntax. Though compact and concise, many compilers have trouble generating efficient code from source written in this notation.

mkmf The mkmf tool may be used to generate Makefiles with correct dependencies for F90 and hybrid-language codes. This places minor restrictions on `module` and `use` statements: these declarations must be on a single line, and the use of continuation lines, e.g:

```
module &
  module_name
use &
  module_name
```

is forbidden.

Use statement 1. The `use` statement must appear on the same line as the module name, i.e, do not use:

```
use &
  module_name
```

This is to be consistent with the dependency analysis performed by `mkmf` outlined above.

2. The `use, only:` clause is *required* so that all imported elements are explicitly declared.
3. Variables imported by a `use` statement must not be modified by the importing module.
4. Modules cannot publish variables and interfaces imported from another module. Thus, each public element of a module is only available through that module.

7.18.2 Style Rules

Style is somewhat personal, and it would be needlessly restrictive to attempt to impose style requirements. These are recommendations which we believe will lead to pleasant encounters with clear, legible and understandable code. The only style requirement we place is that of *consistency*: a single code unit is required to be rigorous in using the author's preferred set of stylistic attributes. It is not onerous to follow a style: modern editors have many language-aware features designed to produce a consistent, customizable style.

Style recommendations include the following:

1. The use of free format;
2. The use of do...end do constructs (as opposed to numbered loops as in Fortran-IV);
3. The use of proper indentation of loops and blocks;
4. The liberal use of blank lines to delimit code blocks;

5. The use of comment lines of dashes or dots to delimit procedures;
6. The use of useful descriptive names for physically meaningful variables; short conventional names for iterators (e.g `(i,j,k)` for spatial grid indices);
7. The use of uppercase for constants (parameters), lowercase for variables;
8. The use of verbose syntax on `end` statements (e.g `subroutine sub...end subroutine sub` rather than `subroutine sub...end`);
9. The use of short comments on the same line to identify variables; longer comments in well-delineated blocks to describe what a portion of code is doing;
10. Compact code units: long procedures should be split up if possible. 200 lines is a rule-of-thumb procedure length limit.

7.19 Lang: C/C++ Coding Standard

- **Use of namespaces** To avoid conflict in symbols, ESMF C++ code will utilize namespaces to add either ESMC (interface code) or ESMCI to symbol names.

  Example:

  The header file for an *ESMC_Widget* is as follows:

```
**file ESMCI_Widget.h
// Internal Widget esmf class
#ifndef ESMCI_Widget_h
#define ESMCI_Widget_h

namespace ESMCI {

class Widget {
public:
Widget();
~Widget();
void Manufacture();
};

} // namespace ESMCI
```

  The implementation file is:

```
***file ESMCI_Widget.C
#include <ESMCI_Widget.h>

namespace ESMCI {

Widget::Widget() {
...
}
Widget::~Widget() {
...
}
```

```
void Widget::Manufacture() {
...
}

} // namespace ESMCI
```

Lastly, when this object is used from C interface code, the following constructs are used:

```
*** file ESMC_SomeInterface_F.C
#include <ESMCI_Widget.h>

extern "C" {
// this cannot be in the ESMCI namespace, else it would change the
// linkage. We have two choices: 1) add using namespace ESMCI and
// then use Widget, else 2) qualify Widget, ESMCI::Widget
void FTN(c_esmc_someinterface)(
ESMCI::Widget **wptr, int *rc){
#undef ESMC_METHOD
#define ESMC_METHOD "c_esmc_someinterface()"
if (rc!=NULL)
*rc = ESMC_RC_NOT_IMPL;

.....
(*wptr)->Manufacture();

}

} // extern "C"
```

- **Protex Prologues** Each C or C++ function, subroutine, or module will include a prologue instrumented for use with the ProTeX auto-documentation script. Prologue templates are included in the ESMF document templates package described in Section 7.1. This convention may be relaxed when absorbing large external code bases that may have their own documentation conventions.

7.20 Repo: Source Code Naming and Tagging Conventions

We provide two types of releases, public and internal, with similar tagging conventions.

7.20.1 Public Releases

Public releases are each given a branch created from the main CVS repository ESMF trunk. The tagging convention for public releases is `ESMF_*_*_*r[p#]`, e.g., `ESMF_0_2_1r`, where the first digit represents a major release, the second digit an incremental release, the third digit a routine update, and an official release `r`. Subsequent patches to the release are identified with the letter `p` followed by the patch number, e.g., `ESMF_1_0_0rp2`. Patches are tagged on the release branch.

7.20.2 Internal Releases

ESMF software internal releases are identified as tags on the CVS main trunk. The tagging convention for internal releases is `ESMF_*_*_*`, e.g., `ESMF_1_0_1`, where the first digit represents a major release, the second digit an incremental release, and the third digit a routine update.

7.21 Data Management Conventions

The ESMF team will adopt the Climate and Forecast (CF) Metadata conventions. These are available at:
http://www.unidata.ucar.edu/software/netcdf/conventions.html

8 Tracking and Metrics

8.1 Release Schedule

The ESMF Release Schedule is generated by the Change Review Board (CRB) (see section 2.3) as the main outcome of its quarterly meetings. The release schedule is posted on the ESMF website home page under **Quick Links**.

8.2 Task List

The Task List is used to archive past, current, and future development tasks. It's updated by the Core Team Manager during and after quarterly CRB meetings. The CRB bases the Release Schedule on the Task List. The Task List is browsable by anyone and is located at http://www.sourceforge.net/projects/esmf (click on Tasks).

8.3 Trackers

SourceForge provides the development team with a set of trackers. Current trackers are Support Requests, Bugs, Feature Requests, Application Issues, and Operations. These are described in more detail in section 8.3.7.

An item entered into a tracker is called a **ticket**. Tickets are assigned a unique number and can move from tracker to tracker. Tickets may be opened in response to a customer request or as a result of some finding or need of the development team.

The process for handling customer requests is located in Section 4.5.

All trackers categorize requests similarly. Pull down menus can assign various criteria including Assignee, Status (Open, Closed, etc.), Category (Array, Regrid, etc.), and Group (Bug, Feature Request, etc.) Tickets can be sorted and viewed based on these criteria and can be moved from one tracker to another. For example a ticket may start out as a support request and be moved to the bug tracker for resolution. To move a ticket, simply change the type using the type pull down menu.

8.3.1 Setting Ticket Priorities

The SourceForge trackers allow the user to set the priority of a ticket. Priority nine is used to label tickets that are associated with a task scheduled for release on the Task List. These are the only tickets that should be labeled at this level. Priority levels one through five can be used by individual developers to prioritize tickets that are not associated with any upcoming release.

8.3.2 Labeling Tickets Longer than 2 Weeks

The CRB only manages tickets that will take longer than two calendar weeks. All tickets that are estimated to be two calendar weeks or longer should be labeled with **LONG:** prepended to the title of the ticket.

Examples:

- **LONG:** wants one-sided communications options (2)
- **LONG:** typekind vs type and kind (4)

8.3.3 Estimating Ticket Completion Time

The following guidelines should be used when determining ticket completion times:

- Estimates are in calendar weeks.
- Estimates do not include the 25% time spent on support. Thus time for ticket items is allocated at 3 weeks per month.
- Estimates do not include integration and release preparation.
- Estimates do include any research time that's part of the developer's position. Calendar weeks are not corrected for this.
- Estimates include design time.
- Design estimates do include looking at other packages, design, design documentation, time for iterating on telecoms, design reviews, feedback, and prototyping.
- Implementation estimates do include implementation, code reviews, testing, and documentation. As a rule of thumb, documentation time and testing time are at least equal to coding time. Also include any peripheral capabilities that will be needed to support the main item.
- Use the past as a guide and estimate long.

8.3.4 Labeling Tickets with Time Estimates

An estimated time to completion should be placed at the end of any ticket title labeled with the **LONG:** key. If the time includes a design phase, this should be included in the total estimate e.g. (7) equals 3 weeks design plus 4 additional weeks for completion.

8.3.5 Cross-Referencing the Task List and Trackers

The tasks in the Task List are usually associated with one or more items in the Feature Request, Bug, and Support Request trackers. In order to cross-reference items, developers should label both the item on the Task List and tracker tickets with a unique key. The key should describe the task, be unique, and be followed by a colon. Careful placement of keys will allow the CRB to quickly visualize through the search mechanism all tasks necessary for a new release. The key is highlighted in the example below:

8.3.6 Summary

There are multiple steps to properly identifying tickets. The preceding sections cover these steps, which are summarized here:

1. Use key words to link tracker tickets with items in the Task List
2. Label all tickets with **LONG:** that will take longer than two calendar weeks
3. Put the completion time estimate after the title (DESIGN + COMPLETION)
4. Label tickets scheduled for release on the Task List with a priority nine.
5. Prioritize other tickets using levels one through five.

In Task List: **STDIZE_INIT**: Standardize Initialization Behavior

In Bug Tracker:

- LONG:**STDIZE_INIT**:Std hand. of Verify rc for declared obj (4)
- LONG:**STDIZE_INIT**:Field count from uninitialized FieldBundle(4)
- **STDIZE_INIT**:Fld ct on uninit Bndle is not 0 on nag (4)
- LONG:**STDIZE_INIT**:BndleGetFildNmes crashes if Bndle uninit(4)

8.3.7 Types of Trackers

- **Support Requests:** Support requests are monitored on the Support Request tracker on SourceForge. (See Support Requests Tracker).
- **Bugs:** Bugs are tracked using the Bug Tracker. Bugs are issues with capabilities that already exist e.g. optimization, documentation, and tests. Bugs can be of short or long duration. Bugs may be identified independently, or they may originate as a support request. If the bug originated independently, the Integrator will create the initial entry. If it originated as a support request, then the **Handler or Advocate** (see section 2.1.1 for definitions) will transfer the support request to the bug tracker. When a bug is opened, provide as much detail as possible to help the Handler reproduce the problem. Indicate where the bug was found e.g. trunk, branch, or both. A bug that exists on both the trunk and a branch will generally be fixed only on the trunk. If the bug is fixed on both the trunk and branch, the ticket must state this explicitly. When the bug is fixed, the Handler adds a note to the ticket and changes the status to Pending. Once the customer who originated the bug or initial support request has confirmed the fix, the ticket can be closed.
- **Feature Requests:** Features are various requests that involve creating new functionality. Features are tracked using the Feature Requests Tracker
- **Applications Issues:** The Applications Issues tracker exists to segregate tickets that involve ESMF code that is embedded in third-party applications and which it is believed the problem lies with the application and not ESMF. This category includes issues with ESMF code that has been modified by a third party, code that has ESMF interface names but was not developed by the Core Team, and and implementation issues with the Application itself. This segregation is necessary because such tickets can remain open for extended periods of time, which would have a negative impact on our support metrics. This tracker is located at Applications Issues Tracker
- **Operations:** Current infrastructure related items are recorded using the Operations Tracker

8.4 Metrics

The Integrator is responsible for tracking various aspects of the ESMF project to measure the implementation and testing progress.

8.4.1 Unit and System Tests Coverage

Throughout the ESMF software development process, the software is constantly being unit and system tested. It is essential to know the percentage of the code that has been fully or partially tested. A script has been written that lists all the public interface subprograms (functions and subroutines) and determines which of these have been unit and/or system tested. From this script output, the tested percentage can be calculated.

8.4.2 ESMF Requirements Coverage

An Excel spreadsheet is maintained listing the requirements as described in the ESMF Requirements Document. A hard copy of the spreadsheet has been posted in a convenient place for the core team. Using color coding the core team indicates which requirements have been implemented and of these which have been tested.

8.4.3 Source Lines of Code, SLOC

The ESMF SLOC information is provided in an internal ESMF web page. The SLOC data is presented in graph form with the following columns:

Fortran

C++

c

Makefiles

SLOC Total

Lines of text

The graphs are a monthly breakdown from January 1, 2002 to the present.

9 Policies

9.1 External Software Libraries

ESMF developers are encouraged to use as few external software libraries as possible, to minimize the effort needed to build and run the ESMF software. The external utilities and libraries that the ESMF distribution relies on are listed on the **Download** tab of the ESMF website.

9.2 Graphics Packages

ESMF uses the MS Visio package to produce its graphics. Both Visio files and *.eps files are checked into the main repository.

Appendix A: Testing Terminology

Industry-accepted definitions exist for software errors and defects (faults), found in the ANSI/IEEE, Glossary of Software Engineering Terminology, are listed in Table A1.

Table A1 – IEEE Software Engineering Terminology [1]

Category	**Definition**
Error	The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.
Fault	An incorrect step, process, or data definition in a computer program.
Debug	To detect, locate, and correct faults in a computer program.
Failure	The inability of a system or component to perform its required functions within specified performance requirements. It is manifested as a fault.
Testing	The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items.
Static analysis	The process of evaluating a system or component based on its form, structure, content, or documentation.
Dynamic analysis	The process of evaluating a system or component based on its behavior during execution.
Correctness	• The degree to which a system or component is free from faults in its specification, design, and implementation. • The degree to which software, documentation, or other items meet specified requirements. • The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not.
Verification	• The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. • Formal proof of program correctness.
Validation	The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

References

[1] IEEE Standard Glossary of Software Engineering Terminology. Technical Report IEEE Std 610.12-1990, Institute of Electrical and Electronic Engineers, December 1990.

[2] Balaji, V. The FMS Manual: A developer's guide to the GFDL Flexible Modeling System. http://www.gfdl.noaa.gov/fms, 2002.

[3] da Silva, A. and Sawyer, W. Protex Distribution (Version 2.0). http://polar.gsfc.nasa.gov/software/protex/protex.php, 2001.

[4] ESMF Joint Specification Team. Earth System Modeling Framework Requirements Document. http://www.esmf.ucar.edu, 2002.

[5] ESMF Joint Specification Team. Earth System Modeling Framework Project Plan. http://www.esmf.ucar.edu, 2005.

[6] ISO/IEC. *The Fortran-95 Standard*. ISO/IEC 1539-1, 1997. Information technology–Programming languages–Fortran–Part 1: Base Language.

[7] Kauffman, B., Bettge, T., Buja, L., Craig, A., DeLuca, C, Eaton, B., Hecht, M., Kluzek, E., Rosinski, J., and Vertenstein, M. http://www.ccsm.ucar.edu, 2001.

[8] McConnell, S. *Rapid Development*. Redmond, Wash.: Microsoft Press, 1996.

[9] Neckels, D. and C. DeLuca. Earth System Modeling Framework Implementation Report. http://www.esmf.ucar.edu, 2002.

Class	Type	Fortran	ESMF
ESMF_Alarm	Deep	public	public
ESMF_AlarmList_Flag	Parameter	public	public
ESMF_Array	Deep		
ESMF_ArrayBundle	Deep		
ESMF_ArraySpec	Shallow		
ESMF_Attribute	Shallow		
ESMF_Base	Deep		
ESMF_BaseTime	Shallow		
ESMF_CWrap	Deep		
ESMF_Calendar	Deep		
ESMF_CalKind_Flag	Parameter		
ESMF_Clock	Deep		
ESMF_CommHandle	Deep		
ESMF_CommTable	Deep		
ESMF_CompClass	Deep		
ESMF_CompType_Flag	Parameter		
ESMF_Config	Deep		
ESMF_ConfigAttrUsed	Shallow		
ESMF_Context_Flag	Parameter		
ESMF_CoordOrder	Parameter		
ESMF_CplComp	Deep		
ESMF_DELayout	Deep		
ESMF_DataHolder	Shallow		
ESMF_DataValue	Ignore		
ESMF_Decomp_Flag	Parameter		
ESMF_Direction_Flag	Parameter		
ESMF_DistGrid	Deep		
ESMF_Field	Deep		
ESMF_FieldBundle	Deep		
ESMF_FieldBundleDataMap	Shallow		
ESMF_FieldBundleType	Deep		
ESMF_FieldDataMap	Shallow		
ESMF_FieldType	Deep		
ESMF_Fraction	Shallow		
ESMF_Grid	Deep		
ESMF_GridComp	Deep		
ESMF_GridStatus_Flag	Parameter		
ESMF_Index_Flag	Parameter		
ESMF_InterfaceInt	Ignore		

Class	Type	Fortran	ESMF
ESMF_LOGENTRY	Shallow		
ESMF_LocalArray	Deep		
ESMF_Log	Shallow		
ESMF_LogKind_Flag	Parameter		
ESMF_Logical	Ignore		
ESMF_Mask	Shallow		
ESMF_LogMsg_Flag	Parameter		
ESMF_NeededFlag	Parameter		
ESMF_ObjectID	Shallow		
ESMF_Pin_Flag	Parameter		
ESMF_PhysGrid	Deep		
ESMF_Pointer	Ignore		
ESMF_ReadyFlag	Parameter		
ESMF_Reduce_Flag	Parameter		
ESMF_Region_Flag	Parameter		
ESMF_RegridMethod_Flag	Parameter		
ESMF_RelLoc	Parameter		
ESMF_ReqForRestartFlag	Parameter		
ESMF_Route	Deep		
ESMF_RouteHandle	Deep		
ESMF_StaggerLoc	Shallow		
ESMF_State	Deep		
ESMF_StateClass	Deep		
ESMF_StateItem	Shallow		
ESMF_StateItem_Flag	Parameter		
ESMF_StateIntent_Flag	Parameter		
ESMF_Status	Parameter		
ESMF_Sync_Flag	Parameter		
ESMF_End_Flag	Parameter		
ESMF_Time	Shallow		
ESMF_TimeInterval	Shallow		
ESMF_VM	Deep		
ESMF_VMId	Deep		
ESMF_VMPlan	Deep		
ESMF_ValidFlag	Parameter		