Community Climate System Model
Software Developer’s Guide

June 2001

Brian Kauffman
Tom Betteg
Lawrence Buja
Tony Craig
Cecelia DeLuca
Brian Eaton
Matthew Hecht
Erik Kluzek
Jim Rosinski
Mariana Vertenstein

Community Climate System Model
National Center for Atmospheric Research, Boulder, CO
http://www.ccsm.ucar.edu/models
CVS tag $Name: $ Build date: August 7, 2002
## Contents

1 Introduction 4

2 Software Management 4
   2.1 Software Management Positions at NCAR 4

3 Software Life Cycle 5
   3.1 The Staged Delivery Model 5

4 Document Guidelines 6
   4.1 General Guidelines 6
       4.1.1 Accessibility 6
       4.1.2 File format 6
   4.2 Development Documents and User Documentation 6
       4.2.1 Requirements Document 6
       4.2.2 Architecture Document 7
       4.2.3 Detailed Design Document 7
       4.2.4 User’s Guide 7
       4.2.5 User’s Reference Manual 7
   4.3 Planning Documents 8
       4.3.1 CCSM Software Engineering Plan 8
       4.3.2 Component Development Plan 8

5 Target Architectures 8
   5.1 Statement from the CCSM Software Engineering Plan 8

6 Coding Conventions 9
   6.1 Fortran Coding Standard for the Community Atmospheric Model 9
       6.1.1 Style Rules 9
       6.1.2 Content Rules 13
       6.1.3 Package Coding Rules 16

7 Configuration Management 19
   7.1 CVS Repository Access 19
   7.2 Procedures for the Central CCSM Repository 19
       7.2.1 Commit Sequence 19
       7.2.2 How to Commit 20
       7.2.3 Useful Modules 21
       7.2.4 Management of Change Logs 21
       7.2.5 Naming Conventions 23

8 Shared and External Libraries 25
   8.1 csm_share 25
       8.1.1 Share Constant Module 25
   8.2 External Libraries Integration Policies 25

9 Component Requirements 26
   9.1 Overall Requirements 26
   9.2 Signal Trapping 26
   9.3 Restart Capabilities 27
       9.3.1 Initial Runs 27
9.3.2 Continue Runs ................................................................. 28
9.3.3 Branch Runs ................................................................. 28

10 Diagnostics ................................................................. 28
10.1 Standard Diagnostics ...................................................... 29
  10.1.1 Timing Calls .............................................................. 29
10.2 Run-Time Optional Diagnostics ......................................... 29
  10.2.1 Global Fluxes ........................................................... 29

11 Data Management Conventions ........................................... 30
  11.1 Overview ......................................................................... 30
    11.1.1 Convention Purpose and Philosophy ......................... 30
    11.1.2 Summary .................................................................... 30
    11.1.3 Relationship to the COARDS Convention .................. 31
11.2 Conventions for Required Attributes ................................ 31
  11.2.1 long_name Attribute .................................................... 31
  11.2.2 units Attribute ............................................................ 31
  11.2.3 Coordinates ............................................................... 31
  11.2.4 Global Attributes ......................................................... 36
11.3 Conventions for Optional Attributes ................................. 36
  11.3.1 Calendar / Orbital Parameters ..................................... 36
  11.3.2 Representing Values on Intervals ................................. 37
  11.3.3 Non-Ordinal Coordinates ............................................ 39
  11.3.4 Projection Coordinates ............................................... 39
  11.3.5 Flux Direction ............................................................ 40
11.4 Resources ......................................................................... 40

12 Component Model Testing, Unit-Testing, Code Reviews .......... 40
  12.1 Component Model Testing ................................................ 41
    12.1.1 Designing Good Tests ............................................... 41
    12.1.2 unit-tests ................................................................. 41
    12.1.3 Functional-tests ........................................................ 42
    12.1.4 System-tests ............................................................. 42
    12.1.5 CCSM Testing requirements and implementation details . 42
12.2 Code-Reviews .................................................................... 43
  12.2.1 Strategies for Implementation of Code-Reviews ............. 43

13 System Testing and Validation ............................................ 43
  13.1 Model Testing Procedures for the CCSM ......................... 44
    13.1.1 Development Testing Steps ....................................... 44
    13.1.2 Ongoing Test Steps .................................................... 45
13.2 Model Validation Procedures for the CCSM ...................... 45
13.3 Port Validation of the CCSM ............................................. 45
  13.3.1 Background ............................................................... 45
  13.3.2 Full CCSM Port Validation .......................................... 46
  13.3.3 Recommended Procedure ............................................ 46
  13.3.4 Port Validation Examples ............................................ 47

Appendix A: Testing Terminology ............................................ 53

Appendix DevA: Component Liaisons ...................................... 54
Appendix DevB: CVS Access Groups 55

Bibliography 56
1 Introduction

This CCSM Software Developer’s Guide is the reference handbook that describes the development practices, standards, and conventions recommended for Community Climate System Model software.

The scope of this document includes any issue related to process, conventions, and standards in the design, implementation, and documentation of any software that is, or is being considered for, use by the CCSM community. Some component-specific guidelines (e.g. Community Atmospheric Model Coding Conventions) are also included while other component-specific guidelines are not included and are deferred to that component’s development team.

This is an evolving document. Suggestions on how to improve the Guide should be sent to csm@ucar.edu.

2 Software Management

Code development teams should clearly identify team leads and other key contact persons. The goal is to establish unambiguous responsibilities and lines of communication with respect to orchestrating the development of the coupled model.

2.1 Software Management Positions at NCAR

The assembly and validation of the evolving coupled CCSM is coordinated by a core group of developers at NCAR, the CCSM Software Engineering Group (CSEG). This team has identified a number of key management roles, which are described below. The individuals who currently fill these roles are identified in Appendix DevA.

CSEG Manager The CSEG manager orchestrates the development of the coupled model. Responsibilities include:

- assembling and tagging working versions of the CCSM modeling system;
- prioritizing and coordinating work on components;
- integrating software developed by the various software development teams;
- working with the component model liaisons to decide when new versions should be produced.

CSEG Quality Assurance Lead The CSEG quality assurance lead system tests new versions of the CCSM system. Responsibilities include deciding the extent of testing needed, facilitating the development of test scripts, and coordinating the actual testing.

CCSM Component Liaisons The CCSM component liaisons monitor development of CCSM components. Their responsibilities include:

- serving as the point-of-contact between the CSEG manager and the component’s working group and/or code development team;
- determining, based on consultation with scientists and other developers, when changes are important enough to warrant a new version;
- recommending to the CSEG manager that a new CCSM version be created;
- maintaining the setup scripts for their component;
- ensuring that a component continues to work in the coupled framework;
- ensuring that work required for compatibility with the fully coupled system is performed.
3 Software Life Cycle

A life cycle is the sequence in which a project specifies, prototypes, designs, implements, tests, and maintains a piece of software. Explicit recognition of a life cycle encourages development teams to address development issues at the appropriate time; for example, to establish basic software requirements before design or coding begins. We recommend that developers roughly follow the staged delivery model (below) when designing significantly new versions of the full model and when developing large components and libraries.

3.1 The Staged Delivery Model

The staged delivery model involves the following steps [5]:

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. Both scientists and the code development team should review and approve the requirements document.

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.

   a. **Detailed Design**
      Create a detailed design document and API specification. This can be done by writing code headers instrumented for ProTeX. Incorporate the interface specification into a detailed design document and review the design.

   b. **Code Construction and Unit Testing**
      Implement the interface, debug and unit test.

   c. **System Testing**
      Assemble the complete system, verify that the code satisfies all requirements.

   d. **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.

Small, simple pieces of software may not require reviews and separate documents at each stage, but it is still a good idea to prepare at least a design document and review it before implementation. Section 4 contains document templates. Since the documents for each successive stage build on each other, a design document contains brief summaries of requirements and system architecture.
4 Document Guidelines

The purpose of these document guidelines is to create a coherent set of documents that clarify the goals of the various software development teams and the functionality of the software they create. These documents are meant to be readily available not only to the development team members, but also to management and other interested parties.

The creation of these documents is closely related to the software life cycle described in Section 3. For example, the existence of a reviewed requirements document demonstrates that the requirements definition phase of a project has been completed.

Section 4.2 provides specific guidelines for each of the documents, suggests responsibility for their creation and maintenance, and describes how the documents are interrelated.

4.1 General Guidelines

4.1.1 Accessibility

Documents developed for the CCSM will be organized and made available for browsing and downloading via the Models and Documentation subsection of the CCSM web site. Eventually they will also be made available to developers through the CCSM repository.

4.1.2 File format

Documents should be available in both web-browsable (e.g. html) and print-friendly formats. We recommend that they be written in LaTeX and based on a set of templates (see section 4.2). 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.

4.2 Development Documents and User Documentation

Below are descriptions of the recommended development documents with guidelines on content. Templates for the documents are available at:

ftp://ftp.cgd.ucar.edu/pub/ccsm/dev_guide/software_docs.tar.gz

Using a consistent format will make it easier to compile the documents and maintain them as a coherent body of documentation.

4.2.1 Requirements Document

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Establishes the functionality required.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>Scientific and computational requirements.</td>
</tr>
<tr>
<td>Notes</td>
<td>Component and module requirements should link to overall project requirements as appropriate.</td>
</tr>
<tr>
<td>Created and Maintained by</td>
<td>In most cases scientists are primarily responsible for defining the requirements of model software. The document may be compiled and maintained by a software engineer who is a lead developer.</td>
</tr>
<tr>
<td>Template</td>
<td>Part of CCSM document template set.</td>
</tr>
<tr>
<td>Example</td>
<td>Coupler 6 Requirements Document</td>
</tr>
</tbody>
</table>
### 4.2.2 Architecture Document

**Purpose**
Describes the overall structure of a software component.

**Contents**
- system overview – a high-level description of the software and its relation to other CCSM software
- reference/link to requirements
- architectural strategies and system architecture
- high level interfaces

**Created and Maintained by**
Lead developer(s).

**Template**
Part of CCSM document template set.

**Example**
Coupler 6 Architecture Document

### 4.2.3 Detailed Design Document

**Purpose**
Used by developers to plan internal and external interfaces before and while the software is being developed.

**Contents**
- API specification, derived types, flag definitions
- description of code function
- reference/link to requirements
- usage examples

**Notes**
We encourage the use of ProTeX to generate the API specification in this document directly from source code.

**Created and Maintained by**
Lead developer(s).

**Template**
Part of CCSM document template set.

**Example**
Time and Date Utility Library

### 4.2.4 User’s Guide

**Purpose**
Explains how to build and run the software.

**Contents**
- a description of the build environment
- a description of any namelist parameters
- a description of any input data
- a description of any output data

**Created and Maintained by**
TBD

**Template**
Part of CCSM document template set.

**Example**
TBD

### 4.2.5 User’s Reference Manual

**Purpose**
Describes the function, interfaces and usage of the software.

**Contents**
API and examples

**Created and Maintained by**
TBD

**Template**
Part of CCSM document template set.

**Example**
TBD
4.3 Planning Documents

4.3.1 CCSM Software Engineering Plan

| Purpose | To provide a high-level overview of CCSM software engineering goals, schedule, progress, and issues. |
| Contents | • executive summary including a statement of the CCSM project’s software objectives  
• high-level directions and strategies for software engineering  
• descriptions of major initiatives related to the CCSM project and their interrelationships, including what groups are involved in code development  
• links to component development plans  
• high-level project schedule and milestones |
| Notes | This Software Engineering Plan contains links to Component Development plans. The Component Development Plans will replace the sections found the "Software Restructuring" section of the current Engineering Plan. |
| Created and Maintained by | Currently maintained by authors. Should be updated before SSC meetings or major releases. The project milestone chart in the CCSM Software Engineering Plan should be updated with Component Development Plans. |
| Template | TBD |
| Example | CCSM Software Engineering Plan |

4.3.2 Component Development Plan

| Purpose | Provide a high-level overview of plans and goals for the component. |
| Contents | Development plan including a schedule with major milestones and the groups involved in development. |
| Notes | These Plans are intended to be a few paragraphs at most. The Software Engineering Plan contains links to the Component Development plans. |
| Created and Maintained by | Component’s liaison to complete system development team. Should be updated before SSC meetings or major releases. The project milestone chart in the CCSM Software Engineering Plan should be updated with Component Development Plans. |
| Template | TBD |
| Example | Coupler 6 Project Plan |

5 Target Architectures

CCSM contains contributions from several different teams of software developers. The goal of this section is to identify specific target architecture requirements so that the various software contributions can be integrated into a system that will run on the desired platforms.

5.1 Statement from the CCSM Software Engineering Plan

This section is excerpted from the CCSM Software Engineering Plan:

"Past CCSM software development has targeted vector platforms and shared memory platforms with relatively modest attention devoted to parallel performance issues. As the availability of vector platforms to U.S. researchers has waned, a different approach is in order. During the period covered by this plan, the main hardware options available to NCAR and the U.S. climate community are likely to be microprocessor-based,
distributed-memory computers and hybrid systems incorporating message passing between shared-memory multiprocessor nodes. Such platforms have complex memory hierarchies, and memory bandwidth and latency issues dominate performance. For the CCSM to meet the goals of the 2000-2005 CCSM Plan, first priority must be placed on making the CCSM run efficiently on these platforms.

Through overseas collaborations CCSM code is in fact likely to be run on both vector and RISC-based machines. Ideally, the code would have the flexibility to run efficiently on both. In practice, this can be difficult to achieve since the choice of optimal data structures, loop ordering, and other significant design decisions may differ depending on whether code is intended for vector or RISC systems.

Since CCSM software engineers do not have access to vector machines as development platforms, it is problematic to consider efficient performance on these platforms as a requirement in the near future. However, it is a priority to write flexible code, and whenever possible, to use data structures that are likely to achieve acceptable performance on either architecture. For excellent performance on a given architecture, this can add considerable complexity, and for some portions of the model, developing two code versions, one scalar and one vector, may be the most practical solution. The availability of development resources may in part dictate the extent to which efficient performance on both types of architectures can be realized.”

6 Coding Conventions

It is recommended that all CCSM components follow a coding convention. The goal is to create code with a consistent look and feel so that it is easier to read and maintain. To date, no conventions have been specified which apply across all CCSM components. Conventions have been defined for the atmospheric model and are included below.

6.1 Fortran Coding Standard for the Community Atmospheric Model

This section defines a set of specifications, rules, and recommendations for the coding of the Community Atmospheric Model (CAM). The purpose is to provide a framework that enables users to easily understand or modify the code, or to port it to new computational environments. In addition, it is hoped that adherence to these guidelines will facilitate the exchange and incorporation of new packages and parameterizations into the model. Other works which influenced the development of this standard are "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).

6.1.1 Style Rules

- **Preprocessor** Where the use of a language preprocessor is required, it will be the C preprocessor (cpp). cpp is available on any UNIX platform, and many Fortran compilers have the ability to run cpp automatically as part of the compilation process. All tokens will be uppercase to distinguish them from Fortran code, which will be in lower case.

- **F90 Standard** CCM4 will adhere to the Fortran 90 language standard. The purpose is to enhance portability, and to allow use of the many desirable new features of the language. If a situation arises in which there is good reason to violate this rule and include Fortran code which is not compliant with the f90 standard, an alternate set of f90-compliant code must be provided. This is normally done through use of a C-preprocessor ifdef construct.

- **Free-Form Source** Free-form source will be used. The f90 standard allows up to 132 characters, but a self-imposed limit of 90 should enhance readability and make life easier for those with bad eyesight, who wish to make overheads of source code, or print source files with two columns per page. The world will not come to an end if someone extends a line of code to column 91, but multi-line comments that extend to column 100 for example would be unacceptable.
- **Loops** Loops should be structured with the do-end do construct as opposed to numbered loops.

- **Argument Comments** Input arguments and local variables will be declared 1 per line, with a comment field expressed with a "!!" character followed by the comment text all on the same line as the declaration. Multiple comment lines describing a single variable are acceptable when necessary. Variables of a like function may be grouped together on a single line. For example:

```fortran
integer :: i, j, k ! Spatial indices
```

- **Continuation Lines** Continuation lines are acceptable on multi-dimensional array declarations which take up many columns. For example:

```fortran
real(r8), dimension(plond,plev), intent(in) :: &
array1, &! array1 is blah blah blah
array2 ! array2 is blah blah blah
```

Note that the f90 standard defines a limit of 39 continuation lines.

Code lines which are continuation lines of assignment statements must begin to the right of the column of the assignment operator. Similarly, continuation lines of subroutine calls and dummy argument lists of subroutine declarations must have the arguments aligned to the right of the "(" character. Examples of each of these constructs are:

```fortran
a - b + c*d + ... + &
h*g + e*f
```

```fortran
call sub76 (x, y, z, w, a, &
b, c, d, e)
```

```fortran
subroutine sub76 (x, y, z, w, a, &
b, c, d, e)
```

- **Indentation** Code within loops and if-blocks will be indented 3 characters for readability.

- **Argument List Format** Routines with large argument lists will contain 5 variables per line. This applies both to the calling routine and the dummy argument list in the routine being called. The purpose is to simplify matching up the arguments between caller and callee. In rare instances in which 5 variables will not fit on a single line, a number smaller than 5 may be used. But the per-line number must remain consistent between caller and callee. An example is:

```fortran
call linemsvc (u3(i1,1,1,j,n3m1), v3(i1,1,1,j,n3m1), t3(i1,1,1,j,n3m1), &
             q3(i1,1,1,j,n3m1), qf cst(i1,1,m,j), xxx)
```

```fortran
subroutine linemsvc (u, v, t, &
                    q, qf cst, xxx)
```

- **Commenting style** Short comments may be included on the same line as executable code using the "!!" character followed by the description. More in-depth comments should be written in the form:

```fortran
! ! Describe what is going on
!```
Key features of this style are 1) it starts with a "!!" in column 1; 2) The text starts in column 3; and 3) the text is offset above and below by a blank comment line. The blank comments could just as well be completely blank lines (i.e. no "!!!") if the developer prefers.

Use of the operators <, >, <-, >-, --, /- is recommended instead of their deprecated counterparts .lt., .gt., .ie., .ge., .eq., and .ne. The motivation is readability.

- **Case** Code will be written in lower case. This convention cleanly segregates code from C preprocessor tokens, since the convention has been established that such tokens are all uppercase.

- **File Format** Embedding multiple routines within a single file and/or module is allowed, encouraged in fact, if any of three conditions hold. First, if routine B is called by routine A and only by routine A, then the two routines may be included in the same file. This construct has the advantage that inlining B into A is often much easier for compilers if both A and B are in the same file. Practical experience with many compilers has shown that inlining when A and B are in different files often is too complicated for most people to consider worthwhile investigating.

The second condition in which it is desirable to put multiple routines in a single file is when they are "CONTAIN"ed in a module for the purpose of providing an implicit interface block. This type of construct is strongly encouraged, as it allows the compiler to perform argument consistency checking across routine boundaries. An example is:

```fortran
file 1:

    subroutine driver
    use mod1
    real :: x, y
    ...
    call sub1(x,y)
    call sub2(y)
    return
    end subroutine

file 2:

    module mod1
    private
    real :: var, var2
    public sub1, sub2

    contains
    subroutine sub1(a,b)
    ...
    return
    end subroutine

    subroutine sub2(a)
    ...
    return
    end subroutine

    end module
```
The number, type, and dimensionality of the arguments passed to sub1 and sub2 are automatically checked by the compiler.

The final reason to store multiple routines and their data in a single module is that the scope of the data defined in the module can be limited to only the routines which are also in the module. This is accomplished with the "private" clause.

If none of the above conditions hold, it is not acceptable to simply glue together a bunch of functions or subroutines in a single file.

- **Module Names** Modules MUST be named the same as the file in which they reside. The reason to enforce this as a hard rule has to do with the fact that dependency rules used by "make" programs are based on file names. For example, if routine A "USE"s module B, then "make" must be told of the dependency relation which requires B to be compiled before A. If one can assume that module B resides in file B.o, building a tool to generate this dependency rule (e.g. A.o: B.o) is quite simple. Put another way, it is difficult (to say nothing of CPU-intensive) to search an entire source tree to find the file in which module B resides for each routine or module which "USE"s B.

  Note that by implication multiple modules are not allowed in a single file.

The use of common blocks is deprecated in Fortran 90 and their continued use in the CCM is strongly discouraged. Modules are a better way to declare static data. Among the advantages of modules is the ability to freely mix data of various types, and to limit access to contained variables through use of the ONLY and PRIVATE clauses.

- **Array Syntax** The use of array syntax is not encouraged. Though compact and concise, many compilers have trouble generating efficient code from source written in this notation. A good general rule is that little or no performance penalty will result from writing (or rewriting) loops containing only a few statements in array syntax. Long, complicated loops containing much work are best coded as explicitly indexed loops.
6.1.2 Content Rules

- **Implicit None** All subroutines and functions will include an "implicit none" statement. Thus all variables must be explicitly typed.

- **Prologues** Each function, subroutine, or module will include a prologue instrumented for use with the ProTeX auto-documentation script (http://dao.gsfc.nasa.gov/software/protex). The purpose is to describe what the code does, possibly referring to external documentation. The prologue formats for functions and subroutines, modules, and header files are shown. In addition to the keywords in these templates, ProTeX also recognizes the following:

```latex
!BUGS:
!SEE ALSO:
!SYSTEM ROUTINES:
!FILES USED:
!REMARKS:
!TO DO:
!CALLING SEQUENCE:
!CALLED FROM:
!LOCAL VARIABLES:
```

These keywords may be used at the developer's discretion.
Prologue for Functions and Subroutines
If the function or subroutine is included in a module, the keyword !IRoutine should be used instead of !Routine.

!------------------------------------------------------------------------------------------------------------------------
! EOP
!
! !Routine: <Function name> (!IRoutine if the function is in a module)
!
! !Interface:
! function <name> (<arguments>)

! !Uses:
! use <module>

! !Return Value:
! implicit none
! <type> :: <name> ! <Return value description>

! !Parameters:
! <type, intent> :: <parameter> ! <Parameter description>

! !Description:
! <Describe the function of the routine and algorithm(s) used in>
! the routine. Include any applicable external references.>
!
! !Revision History:
! YY.MM.DD <Name> <Description of activity>
!
! EOP
!------------------------------------------------------------------------------------------------------------------------!
$Id: code_conv_cam.tex,v 1.3 2001/06/19 21:44:14 kauff Exp$
! $Author: kauff$
!------------------------------------------------------------------------------------------------------------------------
Prologue for a Module

!------------------------------------------------------------------!
! BOP
!
! !MODU LE: <Module name>
!
! !USES:
  use <module>
! !PUBLIC TYPES:
    implicit none
    [save]

    <type declaration>

! !PUBLIC MEMBER FUNCTIONS:
!    <function>            ! Description
!
! !PUBLIC DATA MEMBERS:
    <type> :: <variable>    ! Variable description

! !DESCRIPTION:
! <Describe the function of the module.>
!
! !REVISION HISTORY:
! YY.MM.DD <Name> <Description of activity>
!
! EOP
!------------------------------------------------------------------!
! $Id: code_cony_cam.tex,v 1.3 2001/06/19 21:44:14 kauff Exp $
! $Author: kauff $
!------------------------------------------------------------------!
Prologue for a Header File

!------------------------------------------------------------------------!
! BOP
!
! !INCLUDE:  <Header file name>
!
! !DEFINED PARAMETERS:
!<type> :: <parameter>  ! Parameter description
!
! !DESCRIPTION:
! <Describe the contents of the header file.>
!
! !REVISION HISTORY:
! YY.MM.DD <Name> <Description of activity>
!
! EOP
!------------------------------------------------------------------------!
!
$Id: codeConv_cam.tex,v 1.3 2001/06/19 21:44:14 kauff Exp$
!
$Author: kauff$
!------------------------------------------------------------------------!

- **I/O Error Conditions**: I/O statements which need to check an error condition will use the "iostat=<integer variable>" construct instead of the outmoded **end**- and **err**-. Note that a 0 value means success, a positive value means an error has occurred, and a negative value means the end of record or end of file was encountered.

- **Intent**: All dummy arguments must include the **INTENT** clause in their declaration. This is extremely valuable to someone reading the code, and can be checked by compilers. An example is:

  ```fortran
  subroutine sub1 (x, y, z)
  implicit none
  real(r8), intent(in) :: x
  real(r8), intent(out) :: y
  real(r8), intent(inout) :: z

  y = x
  z = z + x

  return
  end
  ```

6.1.3 Package Coding Rules

The term "package" in the following rules refers to a routine or group of routines which takes a well-defined set of input and produces a well-defined set of output. A package can be large, such as a dynamics package, which computes large scale advection for a single timestep. It can also be relatively small, such as a parameterization to compute the effects of gravity wave drag.

- **Self-containment**: A package should refer only to its own modules and subprograms and to those intrinsic functions included in the Fortran 90 standard. This is crucial to attaining plug-compatibility.
An exception to the rule might occur when a given computation needs to be done in a consistent manner throughout the model. Thus for example a package which requires saturation vapor pressure would be allowed to call a generic routine used elsewhere in the main model code to compute this quantity.

When exceptions to the above rule apply, (i.e. routines are required by a package which are not f90 intrinsics or part of the package itself) the required routines which violate the rule must be specified within the package.

- **Single entry point** A package shall provide separate setup and running procedures, each with a single entry point. All initialization of time-invariant data must be done in the setup procedure and these data must not be altered by the running procedure. This distinction is important when the code is being run in a multitasked environment. For example, constructs of the following form will not work when they are multitasked:

```fortran
subroutine sub
    logical first/.true./
    if (first) then
        first = .false.
        <set time-invariant values>
    end if
```

- **Communication** All communication with the package will be through the argument list or namelist input. The point behind this rule is that packages should not have to know details of the surrounding model data structures, or the names of variables outside of the package. A notable exception to this rule is model resolution parameters. The reason for the exception is to allow compile-time array sizing inside the package. This is often important for efficiency.

- **Precision** Parameterizations should not rely on vendor-supplied flags to supply a default floating point precision or integer size. The f90 "*kind*" feature should be used instead. For example, in CCM4, all routines and modules USE a module named "precision" which defines:

```fortran
integer, parameter :: r8 = selected_real_kind(12)
integer, parameter :: i8 = selected_int_kind(13)
```

Thus, any variable declared real(r8) will be of sufficient size to maintain 12 decimal digits in their mantissa. Likewise, integer variables declared integer(i8) will be able to represent an integer of at least 13 decimal digits. Note that the names r8 and i8 defined above are meant to reflect the size in bytes of variables which are subsequently defined with that "*kind*" value.

- **Bounds checking** All parameterizations must be able to run when a compile-time and/or run-time array bounds checking option is enabled. Thus, constructs of the following form are disallowed:

```fortran
real(r8) :: arr(1)
```

where "arr" is an input argument into which the user wishes to index beyond 1. Use of the (*) construct in array dimensioning to circumvent this problem is forbidden because it effectively disables array bounds checking.

- **Error conditions** When an error condition occurs inside a package, a message describing what went wrong will be printed. The name of the routine in which the error occurred must be included. It is acceptable to terminate execution within a package, but the developer may instead wish to return an error flag through the argument list. If the user wishes to terminate execution within the package,
generic CCM termination routine "endrun" should be called instead of issuing a Fortran "stop". Otherwise a message-passing version of the model could hang. Note that this is an exception to the package coding rule that "A package should refer only to its own modules and subprograms and to those intrinsic functions included in the Fortran 90 standard".

- **Inter-procedural code analysis** Use of a tool to diagnose problems such as array size mismatches, type mismatches, variables which are defined but not used, etc. is strongly encouraged. Flint is one such tool which has proved valuable in this regard. It is not a strict rule that all CCM4 code and packages must be "flint-free", but the developer must be able to provide adequate explanation for why a given coding construct should be retained even though it elicits a complaint from flint. If too many complaints are issued, the diagnostic value of the tool diminishes toward zero.

- **Memory management** The use of dynamic memory allocation is not discouraged because we realize that there are many situations in which run-time array sizing is desirable. However, this type of memory allocation can cause performance problems on some machines, and some debuggers get confused when trying to diagnose the contents of such variables. Therefore, dynamic memory allocation is allowed only "when necessary". The ability to run a code at a different spatial resolution without recompiling is not considered to be an adequate reason to use dynamically allocated arrays.

The preferable mechanism for dynamic memory allocation is automatic arrays, as opposed to **ALLOCATABLE** or **POINTER** arrays for which memory must be explicitly allocated and deallocated. An example of an automatic array is:

```fortran
subroutine sub(n)
   real :: a(n)
   ...
   return
end
```

The same routine using an allocatable array would look like:

```fortran
subroutine sub(n)
   real, allocatable :: a(:)
   allocate(a(n))
   ...
   deallocate(a)
   return
end
```

- **Constants and magic numbers** Magic numbers should be avoided. Physical constants (e.g. pi, gas constants) must NEVER be hardwired into the executable portion of a code. Instead, a mnemonically named variable or parameter should be set to the appropriate value, probably in the setup routine for the package. We realize than many parameterizations rely on empirically derived constants or fudge factors, which are not easy to name. In these cases it is not forbidden to leave such factors coded as magic numbers buried in executable code, but comments should be included referring to the source of the empirical formula.

Hard-coded numbers should never be passed through argument lists. One good reason for this rule is that a compiler flag, which defines a default precision for constants, cannot be guaranteed. Fortran 90 allows specification of the precision of constants through the "_" compile-time operator (e.g. 3.14_r8 or 365_i8). So if you insist on passing a constant through an argument list, you must also include a precision specification. If this is not done, a called routine which declares the resulting dummy argument as, say, real(r8) or 8 bytes, will produce erroneous results if the default floating point precision is 4 byte reals.
7 Configuration Management

It is recommended that all CCSM development teams develop code with the help of a robust software configuration management (SCM) tool, such as CVS. The goal is to establish, document, control, and track the evolution of source code and related documentation throughout the software life cycle.

7.1 CVS Repository Access

The CCSM Repository Access Policy web page describes in greater detail the current access policy and other issues.

To request access to the CCSM code repository for development, please see the CCSM Repository, Request for Access web page. This site contains details about how to request access and the current restrictions on access.

7.2 Procedures for the Central CCSM Repository

This section describes the specific procedures that apply to the central CCSM code repository and official releases of CCSM coupled models.

7.2.1 Commit Sequence

Currently NCAR’s CSEG is using the configuration management tool CVS to manage all of the official CCSM projects. CVS works from a central repository with developers "checking out" a "sandbox" version of a standard release that they can experiment with. When developers have something useful they can then "commit" their changes to the central repository. Other developers get these changes by "updating" their checked out versions with "cvs update". This allows multiple developers to work on changes simultaneously, without having to stumble over file locks. If two developers change the same line of code CVS triggers a "conflict" and displays both versions in the file. The developer then has to pick how to resolve this conflict by hand. CVS also allows a complete configuration of a given project to be tagged with a single version name so that it can be reconstructed later.

CCSM will maintain one coherent repository of all CCSM model codes at NCAR. This repository will contain a module that enables checkout of the complete system (the coupler and all active and data models). CVS online documentation

1. Need for new version driven by global or component issues

The need for a new version of the system will arise either because of project-wide issues or because of changes happening to individual components.

- **CCSM liaison suggests a new version to the CCSM coordinator.** The CCSM liaisons responsible for each component model monitor code changes and decide when changes are large enough to warrant a new CCSM. Any changes to the coupler or csm_share interface – however minor – are Enough to warrant a new CCSM version.

- **CCSM coordinator specifies requirements for a new version.** If global CCSM issues drive the creation of a new version, the CCSM coordinator will work with component model liaisons on a list of requirements for a new CCSM version. CCSM liaisons are responsible for making sure that all necessary changes are implemented.

2. CCSM liaison tests with last CCSM version

The CCSM liaison tests the new component version with the CCSM model system and ensure it works.
3. **Liaison modifies build/run scripts**
The CCSM liaisons are responsible for maintaining the build/run scripts in the CCSM module (the "bld" and "scripts" directories). In general a component model liaison should only work with the scripts for the models they are responsible for. Anyone working with the other scripts should contact the other liaisons, so they know what each is working on. When any commits or tags to the "bld" or "scripts" modules are done the liaison should send mail to the "ccsm-reps@cgd.ucar.edu" e-mail group with a short description of the changes.

4. **Coordinator checks with each component**
The CCSM coordinator checks with the other component liaisons to see if their components need to be updated.

5. **CCSM quality assurance lead tests the system**
The CCSM quality assurance lead tests the entire system. If problems occur in a given component the liaison for that component is responsible for implementing a solution.

6. **CCSM coordinator tags the new version**
The CCSM coordinator tags the complete CCSM model system.

7. **Change Logs documented during tag**
During the tagging procedure the new version is documented and the CCSM web-page automatically updated. Also e-mail is sent out to the "ccsm-progs@cgd.ucar.edu" e-mail group.

8. **New versions checked out to /fs/cgd/ccsm/collections**
The CCSM coordinator then checks out the new CCSM version to /fs/cgd/ccsm/collections. The version is checked out with the same directory name as the tag-name. For example, if the tag-name is "ccsm1\_5\_1" then the version under collections will be checked out as:

   ```
   cd /fs/cgd/ccsm/collections
   cvs co -d ccsm1\_5\_1 -r ccsm1\_5\_1 ccsm1
   ```

**7.2.2 How to Commit**

All access to the CCSM repository is via goldhill. Either run CVS directly on goldhill or via:

:ext:username@goldhill.cg.d.ucar.edu:/fs/cgd/ccsm/models/CVS.REPOS

The later allows a user to access the repository on the CVS server on goldhill while logged into another machine.

`CVSUMASK` and `CVSROOT` must be set properly before check-in.

```
setenv CVSROOT /fs/cgd/ccsm/models/CVS.REPOS
setenv CVSUMASK 002
```

Also set `CVSEDITOR` if you want to use a different editor when entering your commit log. This editor will also be used when editing the ChangeLog rather than the default X11 widgets (for the modules that manage Change-files).
7.2.3 Useful Modules

To list all modules: `cvs co -c`:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVS.Scripts</td>
<td>Scripts for managing the CCSM projects. Miscellaneous tasks such as checking authorization lists, and automatic editing of ChangeLogs when a tag is initiated. (Note: open only to the &quot;egdevadmin&quot; UNIX sub-group, which should be the same as &quot;csm-reps&quot;)</td>
</tr>
<tr>
<td>dev_guide</td>
<td>CCSM Developers Guide (this document).</td>
</tr>
<tr>
<td>bld</td>
<td>CCSM Makefiles.</td>
</tr>
<tr>
<td>csm2</td>
<td>The complete CCSM version 1 project. Contains all official CCSM models, run scripts and Makefiles.</td>
</tr>
<tr>
<td>csm_share</td>
<td>The module of code shared between CCSM components.</td>
</tr>
<tr>
<td>makdep</td>
<td>Dependency generator.</td>
</tr>
<tr>
<td>scripts</td>
<td>CCSM run scripts.</td>
</tr>
<tr>
<td>versions</td>
<td>ChangeLogs for CCSM and several major CCSM component models.</td>
</tr>
<tr>
<td>ccm</td>
<td>Community Climate Model. This module also contains the active land surface model &quot;LSM&quot; and &quot;CLM2&quot;. Directory structure is modified from &quot;csm2&quot;. Several submodules also exist.</td>
</tr>
<tr>
<td>lsm</td>
<td>Active Land Surface Model version 1.</td>
</tr>
<tr>
<td>lsm2</td>
<td>Active Land Surface Model version 2.</td>
</tr>
<tr>
<td>clm2</td>
<td>Community Land Model version 2.</td>
</tr>
<tr>
<td>ccm_lsm_doc</td>
<td>Users-Guide documentation for the CCM and LSM.</td>
</tr>
<tr>
<td>ccm_unit_tests</td>
<td>Unit-testers for ccm source code.</td>
</tr>
<tr>
<td>cpl5</td>
<td>Coupler.</td>
</tr>
<tr>
<td>csim4aio</td>
<td>Community Ice Model and Active Ice model. This module also contains the coupler and necessary data models to run the stand-alone ice model. Directory structure is modified from &quot;csm2&quot;.</td>
</tr>
<tr>
<td>dice5</td>
<td>Data ice model.</td>
</tr>
<tr>
<td>dlnd5</td>
<td>Data land model.</td>
</tr>
<tr>
<td>docn5</td>
<td>Data ocean model.</td>
</tr>
<tr>
<td>latm</td>
<td>&quot;Large&quot; (as in Bill Large) data atmospheric model. Used for running with observational atmospheric datasets rather than model (ie. ccm) datasets.</td>
</tr>
<tr>
<td>pop</td>
<td>Parallel Ocean Model. Active Ocean model. Several submodules also exist.</td>
</tr>
<tr>
<td>timing</td>
<td>General set of timing utilities.</td>
</tr>
<tr>
<td>pilgrim</td>
<td>A generalized set of parallel-decomposition utilities.</td>
</tr>
</tbody>
</table>

7.2.4 Management of Change Logs

The following modules maintain a set of Change-files: csm2, ccm, pop, and csim4aio. In the simplest case there are two files: a ChangeSum and a ChangeLog. These are both kept in the repository for the given component and are accessible from the web. The ChangeSum gives a short one-line synopsis of changes between the current tag and the previous one, while the ChangeLog gives a more detailed description. Some projects also maintain a BranchSum and BranchLog that gives a short or log description of branch tags. These files are maintained outside the repository, but are also web-accessible. POP maintains a LANL_ChangeSum and LANL_ChangeLog kept in the repository to track the changes in LANL versions of POP.

The modules that maintain ChangeLogs use a set of scripts to help make this process easier. These scripts are automatically invoked when a "cvs tag" is invoked. On machines with Perl-Tk installed X11 widgets are used to answer relevant questions. The Change-files are brought up as templates in a simple editor. If
the user wants to use a different editor for the Change-files they need to set the CVSEDITOR environment variable to their selected editor.
7.2.5 Naming Conventions

Major new versions of the CCSM are designated CCSM1.0, CCSM2.0, ... . Significantly new developmental versions are designated CCSM1.1, CCSM1.2, ... . Incremental development versions of the CCSM are designated CCSM2.0-beta01, CCSM2.0-beta02 ... . Development versions represent incremental improvements and developments on the system.
<table>
<thead>
<tr>
<th>Module</th>
<th>Tag</th>
<th>Branch Tag</th>
<th>Frozen Branch Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>bld</td>
<td>bld_[a-b]YYMMDD ex. bld_a001017</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>ccm</td>
<td>ccm_#_# _beta_ ex. ccm1_5 or ccm2_0_beta12</td>
<td>ccm_#_#_branch_[name] ex. ccm1_10_branch_bug_fix</td>
<td>ccm_#_#_branchT_[name]# ex. ccm1_10_branchT_bug_fix1</td>
</tr>
<tr>
<td>cpl</td>
<td>cpl_[a-b]YYMMDD ex. cpl5_a001017</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>ccsm</td>
<td>ccsm_#_# or ccsm_#_#_beta_ ex. ccsm1_5 or ccsm2_0_beta12</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>csim_#_#</td>
<td>csim_#_#_branch_[name]</td>
<td>csim_#_#_branchT_[name]#</td>
<td></td>
</tr>
<tr>
<td>csm_share</td>
<td>share_#_#_branch_name</td>
<td>share_#_#_branchT_[name]_beta_name</td>
<td>share_#_#_branch_name share_#_#_branchT_[name]_beta_name</td>
</tr>
<tr>
<td>dice</td>
<td>dice_[a-b]YYMMDD ex. dice5_a001017</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>dllnd</td>
<td>dllnd_[a-b]YYMMDD ex. dllnd5_a001017</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>docn</td>
<td>docn_[a-b]YYMMDD ex. docn5_a001017</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>latm</td>
<td>latm_[a-b]YYMMDD ex. latm5_a001017</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>pop</td>
<td>POP_#_# _ ex. POP_1_2</td>
<td>POP_#_# _ ex. POP_1_2</td>
<td>POP_#_# _ ex. POP_1_2 _YYYMMDD[a-z] ex. POP_1_2 _20001023a</td>
</tr>
<tr>
<td>scripts</td>
<td>scripts_[a-b]YYMMDD ex. scripts_a001017</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>

Notes: Only "ccsm-reps" can commit changes here.

Branches are used extensively and branch names are tracked in a BranchLog. Frozen releases along branch developments are tracked also in a BranchSum.

Note: Only the CCMS Coordinator can tag new CCMS versions. Currently we are not allowing global CCMS branches. We may maintain bug-fix branches for major CCMS model releases.

Note: Branches are not currently being used, but may be in the future.

Note: Only "ccsm-reps" have the ability to commit changes here. Changes should be made on branches first – to allow full testing before incorporating changes on the trunk. Since, "ccsm\_share" is used by all component models both stand-alone and coupled – it's important that changes here be carefully tested and monitored.

Note: The main trunk is straight LANL POP. The "ccsm\_pop" branches are for local NCAR modifications needed for CCMS releases. Additional branches may be added.

Note: Only "ccsm-reps" can commit changes here.

Notes: Versions with \"[a-b]\" refers to alpha and beta versions of a model release. Versions with \#\_\#\_\# can also be just two digits (for example ccm3\_10) in this case they refer to a major version that has gone through more rigorous testing.
8 Shared and External Libraries

The CCSM has a special csm_share library. This share code was build with ccsm in mind. It has modules that define constants, handle system calls, facilitate component communication, and carry out internal timing among other things.

The CCSM also may use a number of general-purpose software libraries that are or are not widely distributed and in some cases relatively new. Examples include locally developed utility libraries like a timing library or calendar manager as well as externally developed libraries like the Model Coupling Toolkit (MCT) developed at Argonne National Laboratory or the Message Passing Handshaking (MPH) developed at Lawrence Berkeley National Laboratory. It is desirable that all such libraries are integrated into the CCSM in an orderly and consistent fashion. This section describes the policies for integrating such libraries.

8.1 csm_share

csm_share code will be a unique ccsm “component”. The source will be available in the repository similar to all other components and it will be compiled directly into ccsm components as necessary. The requirements for csm_share are

- Must be multi-task and thread safe.
- Must meet general CCSM software engineering standards.
- Must compile and run with all components.
- Source code must be particularly well tested, validated, and documented because of its critical role.

Unit tests should be built for csm_share.

8.1.1 Share Constant Module

When more than one model component uses the same constant, that constant should be defined in the share constant module and all components that use that constant should use the value defined in that module. These can either be manually verified by checking that constants in components agree with the values in the module or by specifically “using” the shr_const_module and defining internal constants from the module directly. The second method is recommended. The share constants values will be reviewed periodically by scientific staff and new constants may be added from time to time.

The values of the share constants are

“to be filled in later”

8.2 External Libraries Integration Policies

For the convenience of users and also for faster modification and debugging in case of problems, we will bundle these library source codes with the CCSM distribution. These libraries will be integrated into the CCSM as follows:

- There will be subdirectories under the ccsm*/utils directory that will contain the different libraries.
- Each library will have its own Makefile or will use the standard CCSM Makefile.
• The library source code and Makefile will be committed into the CCSM repository. Library source code will be updated as necessary when new releases become available. If the codes are kept under revision control elsewhere we will commit versions in which CVS or other directories linked to the remote repositories have been removed.

• Each library will be built into a true lib*.a. The CCSM will link to these libraries. Files in the utils directory will NOT be compiled into CCSM components using an extension to the source path (like csm_share is doing now).

• The csm_share directory will continue to exist and will follow the source path convention for build. At some time in the future, csm_share might be placed under the utils directory and be required to meet the lib*.a standard.

In addition to the above we set certain requirements for external packages. These are:

• Usage of a utility/library must be well documented. In particular, #include or use statements that are required, subroutine/function interfaces, and examples should be provided.

• Unit testers should be provided with the utilities. In general the unit testers will be in a sub-directory beneath the given utility directory called "unit_testers".

9 Component Requirements

All components must meet the following requirements.

9.1 Overall Requirements

Below is an overall list of csm requirements issues. Recommendations can be found in other areas of this document.

• Platforms
• Exact restart for continue, branch
• Branch capability
• Signal Trapping
• Output file naming convention
• Performance

9.2 Signal Trapping

To facilitate system testing and to create more robust CCSM code, all code integrated into the CCSM system must pass the following signal trapping requirements. All code written for the CCSM should be written with these requirements in mind.

Normally not all signal trapping will be on during production runs because of performance issues, but the code must meet these criteria during testing. All codes must pass:

• array bounds checking
• NaN initialization
• underflow to zero
• overflow trap
• divide by zero trap
• invalid operation trap
• inexact operation no_trap (== no requirement)

All of these will be on all the time by default unless some performance degradation is found except
• array bounds checking
• NaN initialization

These will be turned on by default or via an ENV DEBUG flag. The debug flag might also turn on other options to be determined.

On the IBM machines, there are possible conflicts when using both signal trapping and the Mass library. For that reason, use of Mass must be turned off when carrying out signal trapping tests.

9.3 Restart Capabilities

In general, the models should be able to be started from an arbitrary combination of initial or restart data with any start date. The most general cases may require significant user intervention in the scripts. However, the scripts and all components must support the three major run modes as set by ENV variables in the main ccsn script.

These three types of runs available in the CCSM framework are generally known as initial, continue, or branch. Their requirements are described below. The most general ENV variables that specify the run type will be

• CASE : case name
• RUNTYPE : initial, continue, or branch

These variables will be set in the main run script and propagated to the component run scripts.

9.3.1 Initial Runs

Initial runs are specified by setting the RUNTYPE to “initial”. The only requirement for initial runs is that all models must be able to startup from reasonable datasets. These can be generic initial datasets, initial files based on previous model runs, or full precision restart files. There is no dependence of initial runs on previous case names or calendar dates. The initial date of the run will be set by the ENV variable

• BASEDATE : start date for an initial run

REQUIREMENTS:

• All models must be able to startup from reasonable datasets.

and propagated to all components.
9.3.2 Continue Runs

A continue run represents a bit-for-bit exact continuation of a previous run. In this case, it is assumed that the case name is not changed and that the date is continuous. All components must write restart files that will allow them to “restart exactly” under this scenario. In this case, RUNTYPE is set to “continue” and no other changes are required.

REQUIREMENTS:

- All models must be able to restart exactly from the end of any day in the run when compared to a truly continuous run.

9.3.3 Branch Runs

RUNTYPE is set to “branch” to carry out a branch run. A branch run generally means that a case will be started from model restart files with a new case name and possibly a new start date. These types of runs are typically used when sensitivity or parameter studies are required or when runs, like historical or future scenarios, are being carried out.

The simplest scenario of a branch run is to continue a previous case with a new case name and a continuous date. Under this scenario, the new case must produce bit-for-bit exact restart in the same manner as a continue run.

The next scenario of a branch run would be to use a consistent set of restart files but to change both the case name and the start date. Under this scenario, there is no guarantee that the model will produce bit-for-bit exact restart because the date can be non-continuous.

The scripts and components must support the ability change both the case name and start date using ENV variables. The components can pick up the new start date from namelist input (using the ENV variables) or from the coupler at initialization. Exact restart is required if only the case name changes. Under the branch scenario, the primary ENV variables are

- CASE : new case name
- BRANCHDATE : date to branch from
- BRANCHCASE : case to branch from
- RESETDATE : start date of new case

REQUIREMENTS:

- All models must be able to restart exactly from a branch run if only the case name changes when compared to the equivalent continue run.

- The coupler must send the start date to the components at initialization.

- All models must be able to read a restart file from a different case then change both the case name and the start date internally.

10 Diagnostics

All CCSM component models must come pre-instrumented with certain standard timing and flux-conservation diagnostics. These diagnostics are routinely required and it is most efficient that they become part of the component’s base code, rather than being added in an ad hoc fashion during testing.
10.1 Standard Diagnostics

10.1.1 Timing Calls

Five timing calls should be included in every component, and they should be printed at the end of the run. These are

- Timer around the main step loop of the code. This timer represents the total time spent while in the primary "run" loop. Initialization and finalization are ignored.
- Timer around shr_msg_send call. This represents the time spent filling the buffer, calling mpi_send, and waiting.
- Timer around shr_msg_recv call. This represents the time spent calling mpi_recv, waiting, and unpacking the buffer.
- Timer between shr_msg_send and shr_msg_recv in the main loop. This represents the time spent in overlapping work.
- Timer between csm_recv and csm_send in the main loop. This represents the time spent in the other part of the overlapping work.

The sum of the last 4 timers should add up to the first. The timer naming convention or format is not specified at this time. They should be consistent with the component's current style of writing timing output.

10.2 Run-Time Optional Diagnostics

10.2.1 Global Fluxes

Compute the global area integral of each flux before sending and after receiving. These calculations will be used to check the conservation of the fields passed. These changes will probably be added to the csm_send and csm_recv type subroutines. The global area integrals should result in units of Watts, Kg/s and Newtons since the fluxes are sent as W/m², Kg/m²-s, and N/m². These should be written out in full precision and be easily turned on or off through a run time option. We suggest keying this diagnostic off the coupler's info_dbg value which is sent to each component in ibuff(12) and that writing these diagnostics when info_dbg > = 2.

The output format should be written with the following format statement, so it's easy to parse and diff.

\[
\text{write}(6,100) \ 'ocn', 'recv', nfield, global_sum, 'taux'
\]
\[100 \ \text{format}('\text{comm\_diag}', \text{a3}, \text{lx}, \text{a4}, \text{lx}, \text{i3}, \text{es26.19}, \text{a})\]

Where the first variable is a 3 letter component name, the second is "send" or "recv", the third is the field number in the buffer, the fourth is the actual calculated value of the global sum and the fifth is an "optional" string to provide more information than just the field number when looking at the diagnostic.

11 Data Management Conventions

The amount of data generated during CCSM simulations would quickly become unmanageable without establishing data management conventions. The goals of the CCSM data conventions are unifying the format of data files generated by the various CCSM components, cataloging the data generated by numerous CCSM production runs, documenting the contents of a data file, and enabling software tools that display data and perform operations on data files. Also see the CCSM Data Policy at www.cgd.ucar.edu/csm/experiments.
11.1 Overview

CCSM selected netCDF as the standard data format for CCSM related datasets. All CCSM models either create netCDF or provide a filter to convert history files into netCDF. The use of netCDF makes CCSM output data readily accessible to a variety of existing graphics and analysis packages.

Here are the CCSM netCDF Conventions adopted for the output datasets of the CCSM. The convention is designed for the representation of gridded geophysical data. The CCSM netCDF Convention is very similar to the COARDS Conventions, with a few exceptions and additions to meet CCSM requirements.

The CCSM also has Case and File Name conventions to help keep track of the numerous simulations and their output data.

This document contains a description of the netCDF conventions adopted for the output datasets of the NCAR CCSM. The convention is designed for the representation of gridded geophysical data.

11.1.1 Convention Purpose and Philosophy

The netCDF interface enables but does not require the creation of self-describing datasets. The purpose of this convention is to require conforming datasets to contain sufficient metadata that they are self-describing in the sense that each variable in the file has an associated description of what it represents, including physical units if appropriate, and that each value can be located in space and time. Also required is a higher level description pertaining to the variables as a group that gives the source of the data and a history of manipulations that may have been performed on it.

An important potential benefit of a convention is that it can enable software tools that display data and perform operations on specified subsets of the data to do their tasks with minimal user intervention. It is possible using netCDF to provide the metadata describing how a field is located in time and space in many different ways that a human reader would immediately recognize as equivalent. The purpose in restricting how the metadata is represented is to make it practical to write software that allows a machine to be able to figure out what data goes where without need for human intervention. The main restrictions we impose are to require the use of coordinate variables and several variable attributes. No restrictions are placed on the order of a variable’s dimensions.

11.1.2 Summary

The long_name attribute is required for all variables.

The use of coordinate variables is required for all dimensions that correspond to 1D space or time coordinates. Since coordinate variables as defined in section 2.3.1 of the NetCDF User’s Guide are only adequate for describing rectilinear grids, we define a coordinates attribute below to deal with data on irregular grids.

The units attribute is required for all variables used to describe coordinates, and for all other variables for which it is appropriate. The values of the units attributes are character arrays that are recognized by UNIDATA’s udunits package whenever possible. Exceptions are that the units "degree(s)" are not allowed. This convention describes some new values of the units attribute to be used in certain circumstances in which no udunits conformable value exists.

The global attributes title, source, and history are required to provide a minimal description of what the data is, where it came from, and what’s been done to it.

The Conventions attribute is required to identify the location of this document.

In addition to the required features listed above, several conventions have been established to help in the description of gridded data that does not represent pointwise values of the field, but rather represents some characteristic of the field over intervals in space and/or time. A common example of this is time averaged data.
11.1.3 Relationship to the COARDS Convention

This convention has been significantly influenced by the COARDS convention. The differences that may result in files that are not COARDS compliant are the following:

1. COARDS restricts the order of a variable’s dimensions. Because of I/O performance considerations it may not be desirable for all CCSM component models to output their data in conformance with the COARDS requirement. This convention places no restrictions on the order of dimensions.

2. COARDS addresses the issue of dimensionless vertical coordinates, but does not provide a convention that is sufficiently general for our purpose. We would like to use the units attribute to be able to locate the vertical coordinate in space analogously to how the COARDS conventions for latitude and longitude allow one to unambiguously locate points in the horizontal plane. In the case of a dimensionless coordinate some rule is required to convert to one with dimensions. Since there are many possible rules depending on the particular coordinate, we require the value of units to be specific to each coordinate system. Thus we reject "level" and "layer" as being too vague. "sigma_level" describes a particular type of dimensionless coordinate, but not the one used in the atmospheric component of CCSM. Below we provide additional conventions for dimensionless vertical coordinate units attributes.

3. COARDS restricts the names allowed by the netCDF interface by not including the use of the hyphen character, and by recommending that names be case insensitive. We agree that it is better not to use the hyphen character, but feel that case sensitive names should be allowed as is assumed by the netCDF interface.

The NCAR-CCSM convention is more general than COARDS. It is possible to write netCDF files that are compliant with both NCAR-CCSM and COARDS, but NCAR-CCSM conventions don’t restrict files to being COARDS compliant.

11.2 Conventions for Required Attributes

11.2.1 long_name Attribute

The long_name attribute is required for all variables. The value is a character array that contains a long descriptive name.

11.2.2 units Attribute

The units attribute is required for all variables that describe coordinates, and for all other variables whose values represent a dimensional quantity. The values of the units attributes are character arrays that are recognized by udunits whenever possible. The units degree(s) are not allowed (see section 11.2.3 for appropriate units of longitude and latitude). udunits doesn’t have specifications for dimensionless quantities like percent, ppb, and fraction. We recommend that the units attribute still be used, but have not standardized any dimensionless specifiers except for those to indicate vertical coordinates as defined below.

11.2.3 Coordinates

The term coordinate variable is defined in the netCDF User’s Guide to be a 1D array that is used to describe a coordinate in a rectilinear grid. This convention uses the term coordinate variable in that sense. When non-rectilinear grids are used then coordinate information must be represented using higher dimension arrays which do not fit the coordinate variable model. To handle this case the coordinates attribute is introduced below. The term coordinate will be used generically to refer to either the 1D or multidimensional case.

The use of coordinate variables is required for all space or time coordinates that can be represented as 1D arrays.
The values of a coordinate variable must be either strictly increasing or strictly decreasing. The values need not be evenly spaced. Missing values are not permitted. A multidimensional coordinate may use the 
_FillValue or missing_value attributes to indicate grid points that are not used, e.g., in a reduced grid where the number of longitudes on a latitude circle decreases as the latitude gets closer to the poles.

The units attribute is required for all space and time coordinates. The reason for this requirement is that the units attributes can be used by applications (possibly in conjunction with other metadata) to identify what type of coordinate axis corresponds to each of a variable's dimensions. This correspondence does not need to be established by placing restrictions on the dimension ordering.

A. Time

We require the units for a time coordinate to be parsable by udunits as in the following modified excerpt from the udunits documentation:

\texttt{days since 1992-10-8 15:15:42.5 -6:00}

indicates days since October 8th, 1992 at 3 hours, 15 minutes and 42.5 seconds in the afternoon in the time zone which is six hours to the west of Coordinated Universal Time (i.e. Mountain Daylight Time). The time zone specification can also be written without a colon using one or two-digits (indicating hours) or three or four digits (indicating hours and minutes).

The acceptable units for time are listed in the file udunits.dat. The most commonly used of these strings (and their abbreviations) includes day (d), hour (hr, h), minute (min) and second (sec, s). Plural forms are also acceptable. The date string may include date alone; date and time; or date, time, and time zone. The date string is required.

We recommend that the unit year be used with caution. The udunits package defines a year to be exactly 365.242198781 days (the interval between 2 successive passages of the sun through vernal equinox). It is not a calendar year. Udunits includes the following definitions for years: a common\_year is 365 days, a leap\_year is 366 days, a Julian\_year is 365.25 days, and a Gregorian\_year is 365.2425 days.

The calendar calculations done by the udunits package use a mixed Gregorian/Julian calendar, i.e., dates prior to 1582-10-15 are assumed to use the Julian calendar. Time coordinates that use other calendars are thus not able to make use of the udunits library for this purpose. However, it is still required to use the time unit format described above as this contains all the information required to make calendar calculations once the calendar has been specified. We describe a calendar attribute for the time coordinate variable below that may be used for this purpose.

Coordinate variables representing climatological time (an axis of 12 months, 4 seasons, etc. that is located in no particular year) should be encoded like other time axes but with the added restriction that they be encoded to begin in the year 0000. For example,

\texttt{days since 0000-06-15 00:00:0}

indicates days since beginning of the model run starting Jun 15, 0Z.

Time coordinates used for paleoclimate research may involve calendars based on different orbital parameters from those of the present. In this case additional information besides that contained in the calendar attribute may be required. A description of the orbital parameters may be included by using the attribute orbital\_parameters described below. A complete description of the calendar may be included using the global attribute define\_calendar.

B. Dimensional Vertical Coordinates

The acceptable units for dimensional vertical (depth or height) coordinate variables are:

1. Units of pressure as listed in the file udunits.dat. For vertical axes the most commonly used of these include bar, millibar (mbar), decibar (dibar), atmosphere (atm), pascal (Pa), and hPa.
2. Units of length as listed in the file udunits.dat. For vertical axes the most commonly used of these include meter (metre, m), centimeter (cm), decimeter (dm), kilometer (km), and feet (ft).

3. Other units listed in the file udunits.dat that may under certain circumstances reference vertical position such as units of density or temperature.

Plural forms are also acceptable.

The direction of positive (i.e., the direction in which the coordinate is increasing), whether up or down, cannot in all cases be inferred from the units. The direction of positive is useful for applications displaying the data. For this reason the attribute positive as defined in the COARDS convention is required if the vertical axis units are not a valid unit of pressure. Otherwise its inclusion is optional. The positive attribute may have the value up or down (case insensitive).

For example, if an oceanographic netCDF file encodes the depth of the surface as 0 and the depth of 1000 meters as 1000 then the axis would use attributes as follows:

```plaintext
axis_name:units="meters";
axis_name:positive="down";
```

If, on the other hand, the depth of 1000 meters were represented as -1000 then the value of the positive attribute would have been up. If the units attribute value is a valid pressure unit the default value of the positive attribute is down.

Notice that a vertical coordinate variable will be identifiable either by having units of pressure or by the presence of the positive attribute with a value of up or down (case insensitive).

C. Dimensionless Vertical Coordinates

For dimensionless vertical coordinates we introduce conventions designed to facilitate the calculation of the corresponding dimensional coordinates that are required to locate the data spatially. For each of the dimensionless vertical coordinates described below an example of the convention in CDL notation is given for a dimension with the name z.

1. Hybrid Sigma Pressure.

```c
float z(z);
  z:long_name = "hybrid level at layer midpoints";
  z:units = "hybrid_sigma_pressure";
  z:positive = "down";
  z:A_var = "hyam"
  z:B_var = "hybm"
  z:P0_var = "pref"
  z:PS_var = "psurf"
```

A units attribute of hybrid_sigma_pressure means that the pressure at gridpoint \((x(i),y(j),z(k))\) is given by \(p(i,j,k) = A(x) \cdot P0 + B(k) \cdot PS(i,j)\) where the variable names for \(A\), \(B\), \(P0\), and \(PS\) are given by the attributes A_var, B_var, P0_var, and PS_var respectively.

2. Sigma.

```c
float z(z);
  z:long_name = "sigma level at layer midpoints";
  z:units = "sigma_level";
  z:positive = "down";
  z:B_var = "z"
  z:P0_var = "ptop"
  z:PS_var = "psurf"
```
A units attribute of `sigma_level` means that the pressure at gridpoint \((x(i),y(j),z(k))\) is given by
\[ p(i,j,k) = P_0 + B(k) \times (PS(i,j) - P_0) \]
where the variable names for \(B, P_0, \) and \(PS\) are given by the attributes `B_var`, `P0_var`, and `PS_var` respectively.

We recommend that the rule for converting from a dimensionless to a dimensional coordinate be included as a global attribute whose name is same as that of the units specifier with the string `define_` prepended. Here is an example for the CCM3 vertical coordinate:

```c
// global attributes:
   :define_hybrid_sigma_pressure = "\n",
   "Pressure at a grid point (lon(i),lat(j),lev(k)) is computed \n",
   "using the formula: \n",
   "   p(i,j,k) = A(k) \times P_0 + B(k) \times PS(i,j) \n",
   "where A, B, P_0, and PS are contained in the variables whose \n",
   "names are given by the attributes of the vertical coordinate \n",
   "variable A_var, B_var, P0_var, and PS_var respectively. \n",
   "";
```

D. Latitude

The recommended unit of latitude is `degrees_north`. Also acceptable are `degree_north`, `degree_N`, and `degrees_N`.

E. Longitude

The recommended unit of longitude is `degrees_east` (eastward positive). Also acceptable are `degree_east`, `degree_E`, and `degrees_E`. The unit `degrees_west` (westward positive) is not recommended because it implies a negative conversion factor from `degrees_east`.

Longitudes may be represented modulo 360. Thus, for example, -180, 180, and 540 are all valid representations of the International Dateline and 0 and 360 are both valid representations of the Prime Meridian. Note, however, that the sequence of numerical longitude values stored in the netCDF file must be monotonic in a non-modulo sense.

F. Non-Rectilinear Coordinates

When multidimensional variables are required to describe a coordinate, these variables are identified as coordinates by use of the coordinates attribute. The value of the attribute is a string containing the names of variables that describe the coordinate system. There must be at least as many coordinates as there are dimensions for the field variable. There may be more however as in the case of describing a field defined along a trajectory through space.

Examples in CDL notation:

1. Latitude and longitude coordinates both requiring 2D variables:

```c
dimensions:
   nlon = 128 ;
   nlat = 64 ;
   lev = 18 ;
variables:
   float lon(nlat,nlon) ;
   lon:long_name = "longitude" ;
   lon:units = "degrees_east" ;
```
float lat(nlat,nlon);
    lat:long_name = "latitude";
    lat:units = "degrees_north";
float lev(lev);
    lev:long_name = "level";
    lev:units = "mbar";
float T(lev,nlat,nlon);
    T:long_name = "temperature";
    T:units = "K";
    T:coordinates = "lon lat lev";

The coordinates attribute of T tells you that grid point (i,j,k) is located at (lon(i,j),lat(i,j),z(k))
in FORTRAN index conventions). The location of this position in physical space is determined by the interpretations of the coordinates themselves. Because of this there is no restriction on the order in which the coordinate names appear in the coordinate attribute string. Notice that the vertical coordinate is required to use a coordinate variable.

2. Longitude coordinate requiring 2D variable and missing values:

dimensions:
    nlon = 128;
    lat = 64;
variables:
    float lon(lat,nlon);
        lon:long_name = "longitude";
        lon:units = "degrees_east";
        lon:_FillValue = -999.f;
    float lat(lat);
        lat:long_name = "latitude";
        lat:units = "degrees_north";
    float PS(lat,nlon);
        PS:long_name = "surface pressure";
        PS:units = "Pa";
        PS:coordinates = "lon lat";
        PS:_FillValue = 1.0e36f

The 2D longitude coordinate indicates that the longitude values depend on the latitude, and there are missing values. This is the situation when using a grid in which the number of longitude points on a latitude circle decreases going towards the poles.

3. Trajectories:

dimensions:
    time = 1000;
variables:
    float lon(time);
        lon:long_name = "longitude";
        lon:units = "degrees_east";
    float lat(time);
        lat:long_name = "latitude";
        lat:units = "degrees_north";
    float z(time);
        z:long_name = "level";
z:units = "km";
z:positive = "up";

double time(time);
  time:long_name = "time";
  time:units = "days since 1970-01-01 00:00:00";
  time:calendar = "gregorian";

float 03(time);
  03:long_name = "ozone concentration";
  03:units = "ppbv";
  03:coordinates = "lon lat z time";

11.2.4 Global Attributes

The required global attributes are intended to provide information about where the data came from and what has been done to it. This information is mainly for the benefit of human readers. These attributes are all character arrays. For readability in ncdump outputs it is recommended to embed newline characters into the arrays to break them into lines.

title A succinct description of what is in the data set.

source Source of the data. For CCSM data this could be the specific model component that produced the data along with a description of modifications made to the standard model component.

history Contains a line for each invocation of a program and arguments that were used to derive the file. Well-behaved generic netCDF filters will automatically append their name and the parameters with which they were invoked to the global history attribute of an input netCDF file. We recommend that each line begin with a timestamp indicating the time and date that the program was invoked.

conventions A conforming dataset should use the value NCAR-CSM which indicates that the conventions followed by the dataset may be found on the host machine ftp.unidata.ucar.edu in the directory pub/netcdf/Conventions/NCAR-CSM/.

11.3 Conventions for Optional Attributes

11.3.1 Calendar / Orbital Parameters

In order to calculate a new date and time given a base date, base time and a time increment one must know what calendar to use. For this purpose we recommend that the attribute calendar be assigned to time coordinate (or be global) whenever its units are of the form time units since base date, base time. The values currently defined for calendar are:

gregorian Modern calendar. (Default)

noleap Modern calendar without leap years, i.e., all years are 365 days long.

julian Julian calendar.

n kyr B.P. A generic designation for a paleoclimate calendar valid n thousand years before present where n should be replaced by the appropriate value. A precise definition of the calendar may be included by using the global attribute define_calendar.

In paleoclimate research the calendar may be described with reference to an orbit that is different from that of present day. This should be described by using an attribute of the time coordinate (or global) called orbital_parameters to give a list of the names of variables that contain the eccentricity, obliquity, and perihelion data. The method of attaching calendar dates to the orbit can be described using the global attribute define_calendar, for example:
11.3.2 Representing Values on Intervals

It is often the case that data on a grid does not represent the point values of some field variable but instead represents some characteristic of the field (i.e., the result of some mathematical operation performed on the field values) over intervals of space and/or time. Typically this might be a weighted average or perhaps the minimum or maximum values over the intervals. Hence we require methods to describe both the characteristic of the field over intervals and what the intervals are that correspond to the grid points.

To represent the characteristic of the field over intervals we introduce attributes of the form coord_op, where the string coord should be replaced by the actual name of the coordinate, for example lat_op, or time_op. The values of these attributes are character arrays that describe the operation that has been performed on the corresponding coordinate. The currently defined values are:

- "point"  value is at a point (default, does not need to be specified)
- "minimum"  minimum of values over an interval
- "maximum"  maximum of values over an interval
- "sum"  sum of values over an interval
- "average"  average of values over an interval
- "rms"  root mean square of values over an interval
- "range"  difference between maximum and minimum values over an interval

coord_op attributes may be applied to individual variables or may be global which implies that the operation has been applied to all variables which use the indicated coordinate. If coord_op is a global attribute, it may still be applied to a variable to override the value of the global attribute.

To represent the intervals we add the attribute bounds to the appropriate coordinate. The value of bounds is the name of the variable that contains the interval boundaries. In the case where the grid is defined by coordinate variables, and where the intervals are contiguous this can be a variable whose dimension size is one larger than the dimension size of the corresponding coordinate variable. The relationship between a coordinate variable, say x, and the variable that contains the interval bounds, say x_bound, is that the value x(i) is contained in the interval with boundaries x_bound(i) and x_bound(i+1). In the more general case (still 1D) where the intervals may be disjoint or possibly overlapping then the variable containing the bounds is 2D with the 2nd dimension (in C notation) being the same size as the dimension of the corresponding coordinate variable. In this case the relationship is that the value x(i) is contained in the interval with boundaries x_bound(0,i) and x_bound(1,i).
Examples in CDL notation:

1. Time averaged variable, contiguous intervals:

```plaintext
dimensions:
  time_bound - 4 ;
  time - 3 ;
variables:
  double time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1970-01-01 00:00:00" ;
    time:calendar = "gregorian" ;
    time:bounds = "time_bound" ;
  double time_bound(time_bound) ;
    time_bound:long_name = "time interval boundaries" ;
    time_bound:units = "days since 1970-01-01 00:00:00" ;
float gaTS(time) ;
  gaTS:long_name = "global average surface temperature" ;
  gaTS:units = "K" ;
  gaTS:time_op = "average" ;
data:
  time - .25, .5, .75 ;
  time_bound - 0., .25, .5, .75 ;
```

The variable gaTS represents 6 hour time averages. The first time sample corresponds to a time average starting at 1970-01-01 0Z and ending at 1970-01-01 6Z. The values of the time coordinate are arbitrarily set to the end of the corresponding averaging interval.

2. Time averaged variable, disjoint intervals:

```plaintext
dimensions:
  d2 - 2 ;
  time - 3 ;
variables:
  double time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1970-01-01 00:00:00" ;
    time:calendar = "gregorian" ;
    time:bounds = "time_bound" ;
  double time_bound(d2,time) ;
    time_bound:long_name = "time interval boundaries" ;
    time_bound:units = "days since 1970-01-01 00:00:00" ;
float gaTS(time) ;
  gaTS:long_name = "global average surface temperature" ;
  gaTS:units = "K" ;
  gaTS:time_op = "average" ;
data:
  time - 31., 396., 761. ;
  time_bound(0,*) - 0., 365., 730. ;
  time_bound(1,*) - 31., 396., 761. ;
```

11.3.3 Non-Ordinal Coordinates

It is useful to adopt a convention for certain types of dimensions that do not map into netCDF coordinate variables, but that aren’t simply generic ordinal dimensions. For example, we save certain quantities associated with particular islands or ocean basins. Here the coordinate value is the character string containing the island or basin name which cannot be used as a netCDF coordinate. One can use character string variables that have the same name as the dimension with _label attached to associate these values with the corresponding dimensions. e.g.,

dimensions:
  time - UNLIMITED ; // (12 currently)
  z_t - 45 ;
  nchar - 132 ;
  islands - 8 ;
  basins - 8 ;
variables:
  double time(time);
  time:long_name = "time";
  time:units = "days since 0000-00-00 00:00:00";
  float z_t(z_t);
  z_t:long_name = "Depth (T grid)";
  z_t:units = "centimeters";
  z_t:positive = "down";
  char islands_label(islands, nchar);
  islands_label:long_name = "Islands";
  char basins_label(basins, nchar);
  basins_label:long_name = "Ocean Basins";
  float T_horz(time, basins, z_t);
  T_horz:long_name = "Horizontal Average Potential Temperature";
  T_horz:units = "celsius";
  float pisle(time, islands);
  pisle:long_name = "Island Streamfunction";
  pisle:units = "centimeters-3/second";

11.3.4 Projection Coordinates

For grids that are rectilinear in a cartographic projection space the units of the coordinate variables (typically meters or km) are not sufficient to allow one to determine which one corresponds to the abscissa and which to the ordinate. Nor do the units allow one to determine the order in which the coordinates should be used as arguments to a function that returns the corresponding longitude and latitude values. We suggest the attribute proj_coordinates be used to name the coordinate variables that represent the abscissa and ordinate respectively in the cartesian system (order is important here). proj_coordinates can be a variable attribute or a global attribute allowing variable override.

We recommend the USGS software package proj as the reference implementation for cartographic projections. This means that proj defines the parameter interface and transformation behavior for projections, although you are free to use any implementation that agrees with proj. To define the projection, we recommend therefore using the attribute proj_parameters whose value is a string that would be used to describe the projection to the proj software. The proj software is described in a user’s manual that may be viewed by clicking on the highlighted proj string and then on the highlighted string PROJ.4.3.ps.gz.

Example in CDL notation:

Mercator projection, central meridian at 90W.
dimensions:
  x - 128;
y - 64;
z - 18;
time - 100;
variables:
  float x(x);
    x:long_name = "x-coordinate";
    x:units = "km";
  float y(y);
    y:long_name = "y-coordinate";
    y:units = "km";
  float z(z);
    z:long_name = "level";
    z:units = "km";
    z:positive = "up";
  double time(time);
    time:long_name = "time";
    time:units = "days since 1970-01-01 00:00:00";
    time:calendar = "noleap";
  float U(time,z,y,x);
    U:long_name = "zonal wind component";
    U:units = "m/s";
    U:proj_coordinates = "x y";
// global attributes:
  :proj_parameters = "+proj=merc +lon_0=90W";

If the projection is not found in the proj software then use the attribute define_projection to supply a complete description or reference to where a description can be found.

11.3.5 Flux Direction

We recommend making the direction of all flux quantities explicit by the use of the variable attribute flux_direction. The values should be directions such as up, down, north, south, east, or west.

11.4 Resources

NCAR Climate Systems Model
  UNIDATA NetCDF Home Page
  UNIDATA’s NetCDF Conventions Page
  UNIDATA udunits Software
  U.S. Geological Survey proj Software
  Brian Eaton, eaton@ucar.edu

12 Component Model Testing, Unit-Testing, Code Reviews

Complex software such as the CCSM requires extensive testing in order to prevent model defects and to provide stable, solid models to work with. Layered testing has shown to be the most effective in catching software defects. Layered testing refers to testing on different levels, both testing individual subroutines as well as more complex systems. There may be several layers of simple to more complex systems tested as well. Testing the individual component models stand-alone is an example of a system less complex than
the entire CCSM. Unit-testing is the first layer – testing individual subroutines or modules. Unit-testing by itself will not catch defects that are dependent on relationships between different modules – but testing the entire system sometimes will not catch errors within an individual module. That is why using both extremes is useful in catching model defects. Section ?? covers testing for the entire CCSM modeling system, this section goes over testing of individual model components and unit-testing of subroutines and modules within those components. Another way to help eliminate code errors are periodic code-reviews. Code-reviews can be implemented in many different fashions, but in general it involves having at least one person besides the author go through the written code and examine the implementation both for design and errors. Jones-1986[3] states that “the average defect-detection rate is only 25 percent for unit testing, 35 percent for function testing, and 45 percent for integration testing. In contrast, the average effectiveness of design and code inspections are 55 percent and 60 percent. McConnell-1993[4] also notes that as well as being more effective in catching errors, code-reviews also catch different types of errors than testing does. In addition when developers realize their code will be reviewed – they tend to be more careful themselves when programming.

Since, the CCSM and the component models take substantial computer resources to run – catching errors early can cut computing costs significantly. In addition to that as pointed out by McConnell-1993[4] development time decreases dramatically when formal quality assurance methods including code-reviews are implemented.

12.1 Component Model Testing

Each component model needs to develop and maintain it’s own suite of testing for that given component. It is recommended that analysis of the kinds of testing required for each model by each component models development team be done and written down in a formal testing-plan. Also creating automated tests to run a suite of standard tests can be useful to ensure the models work and continue to work as needed. This is especially useful for making sure models continue to work on multiple platforms. McConnell-1996[5] refers to this as the daily “build and smoke test” you daily build and run your code to ensure it continues to work and doesn’t just sit there and “smoke”.

12.1.1 Designing Good Tests

In order to design a comprehensive testing plan we want to take advantage of the following types of tests.

unit-testing Testing done on a single subroutine or module.

functional-testing Testing for a given functional group of subroutines or modules, for example, testing model dynamics alone without the model physics.

system-testing Testing done on the whole system.

12.1.2 unit-tests

Unit-tests are a good way to flush out certain types of defects. Since unit-tests only run on one subroutine they are easier to use, faster to build and run, allow more comprehensive testing on a wider range of input data, help document how to use and check for valid answers, and allows faster testing of individual pieces. By building and maintaining unit-tests the same tests can be run and used by other developers as part of a more comprehensive testing package. Without maintaining unit-tests developers often do less testing than required – since system tests are so much harder to do – or they have to “hack” together their own unit-tests for each change. By maintaining unit-tests we allow others to leverage off previous work and provide a format to quickly do extensive checking.

Good unit-tests will do the following:

1. Applicable requirements are checked.
2. Exercise every line of code.

3. Check that the full range of possible input data works. (i.e. if Temperature is input check that values near both the minimum and maximum possible values work)

4. Boundary analysis – logical statements that refer to threshold states are checked to ensure they are correct.

5. Check for bad input data.

6. Test for scientific validity.

By analyzing the code to be tested different test cases can be designed to ensure that all logical statements are exercised in the unit-test. Similarly input can be designed to test logical threshold states (boundary analysis). Testing scientific validity is of course the most difficult. But, sometimes testing states where the answer is known analytically can be useful. And ensuring (or measuring) the degree to which energy, heat, or mass is conserved for conservative processes can also often be done. These types of tests may also be applied for more complex functional and system tests as well.

12.1.3 Functional-tests

Functional tests take a given sub-set of the system and test this set for a particular functionality. Scientific functional tests are common. For example, the Column Radiation Model (CRM) is used to check the radiation part of the atmospheric model. Quite often scientific functional tests are implemented as name-list options to component models, the atmospheric model has a name-list item to test dynamics only by turning the physics off. Functional tests could also be created for infra-structure issues such as parallel decomposition or handling of input or output data. Important functional tests should be maintained in CVS as separate modules that include the directories maintained for the main component model.

12.1.4 System-tests

System tests for a given component model need to ensure that the given model compiles, builds, and runs and that it passes important model requirements. For example, most models require that restarts give results that are bit-for-bit to continuous simulations. This requirement can be tested fairly easily.

12.1.5 CCSM Testing requirements and implementation details

Unit-tests for component models should meet the following minimum requirements.

1. Maintained in CVS either with the rest of the models source code or as a separate module that can be used. Module and/or directory name should be easily identifiable such as ‘unit_tests’.

2. Have documentation on how to use it.

3. Check error conditions so that an error will print out problems.

4. Prompt for any input in a useful way. (so you don’t have to read the code to figure out you have to enter something).

5. Have a Makefile associated with it. It may be useful to leverage off the main Makefile so that the compiler options are the same and so that platform dependencies don’t have to be maintained twice.

6. In general unit-tests should be run with as many compiler debug options on as possible (bounds checking, signal trapping etc).

Component model system tests should meet the following minimum requirements.
1. Ensure that the given model will compile, build and run on at least one production platform.
2. Ensure that the given model will work with the CCSM system on at least one production platform.

12.2 Code-Reviews

Formal reviews of the code where the code is gone through line-by-line in groups or in pairs has shown to be one of the most effective way to catch errors McConnell-1993 [4]. As such it is recommended that component model development teams create a strategy for regularly reviewing the code.

12.2.1 Strategies for Implementation of Code-Reviews

Code reviews can be implemented in different ways.

- **Code librarian** – Before code is checked into CVS it goes through a “librarian” who not only is responsible for testing, and validation of the changes – but also reviews it for design and following code standards.
- **Peer reviews** – Before code is checked into CVS a peer developer reviews the changes.
- **Pair programming** – All code is developed with two people looking at the same screen (one of the practices of Extreme Programming – [2]).
- **Formal configuration management** – All code modifications are presented to a configuration management team who extensively reviews and tests changes and incorporates changes as dictated by project management.
- **Formal group walk-through** – Code is presented and gone through by an entire group.
- **Formal individual walk-through** – Different individuals are assigned and take responsibility to review different subroutines.

It is recommended that development teams provide both a mechanism to review incremental changes, and also have formal walk-through of important pieces of code in group. This serves two purposes: the design is communicated to a larger group, and the design and implementation is also reviewed by the entire group.

By adopting quality assurance techniques CCSM model codes can both be of greater quality, development time can be lowered, and machine time can be cut by decreasing errors.

13 System Testing and Validation

Regular system testing and validation of the CCSM is required to ensure that model quality and integrity is maintained throughout the development process. This section establishes the system testing standards and the procedures that will be used to verify the standards have been met. It is assumed that component model development teams have unit tested their component prior to making it available for system testing. See section ?? for more information on testing of individual components and unit-testing of individual subroutine and modules within components.

There are two general categories of model evaluations: frequent short test runs and infrequent long validation integrations.

Model **testing** refers to short (3 to 31 day) model runs designed to verify that the underlying mechanics and performance of the coupled model continues to meet specifications. This includes verifying that the model actually starts up and runs, benchmarking model performance and relative speed/cost of each model component as well as checking that the model restarts exactly. These tests are done on each of the target
platforms. Model testing does not address whether the model answer is correct, it merely verifies that it mechanically operates as specified.

Model validation involves longer (at least 1 year) integrations to ensure that the model results are in acceptable agreement with both previous model climate statistics and observed characteristics of the real climate system. Model validation occurs with each minor CCSM version (i.e. CCSM2.1, CCSM2.2) or at the request of the CCSM scientists and working groups. Once requested, model validation is only carried out after CCSM scientists have been consulted and the model testing phase is successfully completed. The model validation results are documented on a publicly assessable web page (http://www.ccss.ucar.edu/models/ccsm2.0beta/testing/status.html).

Port validation is defined as verification that the differences between two otherwise identical model simulations obtained on different machines or using different environments are caused by machine roundoff errors only.

13.1 Model Testing Procedures for the CCSM

Formal testing of the CCSM is required for each tagged version of the model. The CCSM quality assurance lead is responsible for ensuring that these tests are run, either by personally doing it or having them run by a qualified person. If a model component is identified as having a problem, the liaison for that component is expected to make resolving that problem their highest priority. The results of the testing and benchmarking will be included in the tagged model to document the run characteristics of the model. The actual testing and analysis scripts will be part of the CCSM CVS repository to encourage use by outside users.

13.1.1 Development Testing Steps

1. Successful build CCSM shall compile on each of the target platforms with no changes to the scripts, codes or datasets.

2. Successful startup CCSM will start from an initial state and run for 10 days.

3. Successful restart CCSM will start from an initial state and halt after 5 days, then restart and run from day 6 to day 10.

4. Successful branch CCSM will start from an initial state and halt after 5 days, then carry out a branch case with only a case name change and run from day 6 to day 10.

5. Exact restart A bit-for-bit match must occur between the 10-day initial run and the restart run and branch runs using the same number of processors.

6. Signal trapping A signal trapping test should be conducted with the environment variable DEBUG set to true in the Makefile.

7. Other diagnostics A diagnostic test will be performed with info_dbg set to level 2 in the coupler input.

8. Port diagnostics A port diagnostic test will be performed with info_bcheck set to level 3 in the coupler input and a 10 day run will be carried out.

9. Performance benchmarking The total CPU time, memory usage, output volume, GAU cost, disk space use and wall clock time for the 10 day run will be recorded. The relative cost of each component will also be recorded.

10. Test report The results of all steps above are to be documented in a test report with emphasis on results, comparisons to the previous test and recommendations for improvements. Any faults or defects observed shall be noted and must be brought to the attention of the liaison responsible for that component and the software engineering manager.
13.1.2 Ongoing Test Steps

1. *Smoke-test* A major criteria used in evaluating the effectiveness of a test procedure is the length of time which has lapsed since the last time the system was tested. To test for system or software changes, an automated six day test run will be made each weekend with the latest CCSM distribution on each of the supported platforms. A restart test will conducted on the first weekend of each month.

2. *Test report* The results of steps 1 will be automatically documented in a test report.

13.2 Model Validation Procedures for the CCSM

Model Validation occurs with each Minor CCSM version (i.e. CCSM2.1, CCSM2.2) or at the request of the CCSM scientists and working groups. Before starting a validation run, the CCSM Quality Assurance Lead will consult with the CCSM scientists to design the validation experiment.

**Pre-Validation Run Steps:**

1. *Tests successfully* The validation will successfully complete the testing steps outlined above.

2. *Scientist sign-on* The CCSM scientists must agree to make themselves available to informally analyze the results of the run during the run and formally review the results within one week of the completion of the run.

**Validation Steps:**

1. *Comparison with previous model runs* Result agrees with previous model runs

2. *Comparison with observed climate* Result agrees with observed climate

13.3 Port Validation of the CCSM

13.3.1 Background

Port validation is defined as verification that the differences between two otherwise identical model simulations obtained on different machines or using different environments are caused by machine roundoff errors only. Roundoff errors can be caused by using two machines with different internal floating point representation, or by using a different number of processing elements on the same machine which may cause a known re-ordering of some calculations, or by using different compiler versions or options (on a single machine or different machines) which parse internal computations differently.

The following paper offers a primary reference for port validation (hereafter referred to as RW):

As established in RW, three conditions of model solution behavior must be fulfilled to successfully validate a port of atmospheric general circulation models:

1. during the first few timesteps, differences between the original and ported solutions should be within one to two orders of magnitude of machine rounding;

2. during the first few days, growth of the difference between the original and ported solutions should not exceed the growth of an initial perturbation introduced into the lowest-order bits of the original solution;

3. the statistics of a long simulation must be representative of the climate of the model as produced by the original code.

The extent to which these conditions apply to models other than an atmospheric model has not yet been established. Also, note that the third condition is not the focus of this section (see section 13.2).
13.3.2 Full CCSM Port Validation

Validation of the full CCSM system, defined as the combination of all active model components participating in the full computation, is a two-step process:

1. Validate each model as a standalone system
2. Validate the coupled system

Validation of each component model alone should be performed by the model developers, and it may not be necessary to perform the standalone tests as part of regular, frequent validation testing.

To validate the fully coupled CCSM, the objective is to establish a procedure which will allow one to conclude confidently that the port of the full system (all components active) is valid. However, there are at least two potential problems which should be noted:

- Will the procedure be sufficient to draw conclusions confidently? That is, it must have little potential to conclude a good port when the port is, in fact, bad.
- Upon a conclusion that the port is bad, it is likely that no information will be available pinpointing which component of the full system is suspect.

13.3.3 Recommended Procedure

The general procedure for port validation of the full CCSM is to examine the growth of differences between two solutions over a suitable number of integral timesteps. This error growth can be compared to the growth of differences between two solutions on a single machine, where the differing solution was produced by introducing a random perturbation of the smallest amplitude which can be felt by the model at the precision of the machine.

It is recommended that the procedure examine the growth of differences in a state variable which resides at the primary physical interface (that is, the surface), where the accumulation of errors in all components will act quickly and where the action of the CCSM coupler is also significant (for example, grid mapping).

It is also recommended that the procedure be performed on a coupled system where the exchange of information between active components is frequent. Exchanges of information a model day boundaries may mask the detection of an invalid port because the magnitude of the error differences could reach roundoff saturation levels prior to an exchange of data. See example 5 in section 13.3.4.

The recipe for CCSM validation is as follows:

1. run the CCSM on a selected machine on which confidence in the solution has been established;
2. re-run the CCSM on the same machine, introducing an initial error perturbation in the atmospheric model 3-D temperature using the procedure available in the CCM (see -need a web link-);
3. run the CCSM on the target machine using the same code, same model input namelist files, and same model input data files, and compare the error growth in the perturbed solution versus the error growth in the ported solution.

The errors should satisfy the first two conditions described in RW.

Specific recommendations for a port validation of CCSM:

<table>
<thead>
<tr>
<th>Item</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>length of test</td>
<td>5-30 days</td>
</tr>
<tr>
<td>field to examine</td>
<td>2-D surface temperature on atmospheric grid</td>
</tr>
<tr>
<td>frequency of samples</td>
<td>every timestep</td>
</tr>
<tr>
<td>size of perturbation</td>
<td>smallest which can be felt on original machine (1.0E-14)</td>
</tr>
<tr>
<td>error statistic</td>
<td>RMS difference of field, area-averaged</td>
</tr>
</tbody>
</table>
Note that the field being examined must be processed using the full machine precision. The field must be saved at full machine precision during the model history archival step, and the error statistic must be computed at full machine precision.

13.3.4 Port Validation Examples
Example 1. Perturbation Error Growth

A typical perturbation error growth of the globally averaged RMS difference of surface temperature using a control and a low-order bit perturbation of CCM on 16pes of the IBM SP. Two days (144 atmospheric timesteps) are shown. Note that the first few timesteps satisfy the first condition of RW.
Example 2. Machine Port
Black line is the perturbation error growth on the original machine (same as example 1). Red line is the growth in differences between the simulation on the original machine and the simulation on 64pes of an SGI Origin 2000, and the blue line is the growth of differences from a simulation on 32pes of an IBM SP. Note that the first two days (144 timesteps) satisfy the second condition of RW.
Surface Temperature RMS Difference (T42 grid)

Example 3. Bad Port I
Same as example 2, but blue line is a port where the default Greenhouse gas concentration was modified accidentally in the atmospheric source code. The first and second conditions of RW are violated.
Example 4. Bad Port II
Same as example 2, but blue line is a port where the second order diffusion coefficient was raised by 15% in the atmospheric model namelist input. The first and second conditions of RW are violated.
Example 5. Frequency of Model Data Exchange
Same as example 2, but blue line is a port where the ocean model vertical diffusion coefficient was lowered intentionally. While the first and second RW conditions are satisfied, the port was forced to have been bad. The problem is that the ocean and atmosphere were directed to exchange data only at day boundaries (72 atmospheric timesteps), and thus the coupler did not communicate the ocean solution to the atmosphere until the start of the second day. The error in the ocean model solution had already reached the roundoff saturation level by the time the atmospheric model received the information. For port validation, this example demonstrates that the exchanges of data between components must occur more frequently than time scale at which the roundoff error reaches a level (saturated) value.
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]

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error</td>
<td>The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.</td>
</tr>
<tr>
<td>Fault</td>
<td>An incorrect step, process, or data definition in a computer program.</td>
</tr>
<tr>
<td>Debug</td>
<td>To detect, locate, and correct faults in a computer program.</td>
</tr>
<tr>
<td>Failure</td>
<td>The inability of a system or component to perform its required functions within specified performance requirements. It is manifested as a fault.</td>
</tr>
<tr>
<td>Testing</td>
<td>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.</td>
</tr>
<tr>
<td>Static analysis</td>
<td>The process of evaluating a system or component based on its form, structure, content, or documentation.</td>
</tr>
<tr>
<td>Dynamic analysis</td>
<td>The process of evaluating a system or component based on its behavior during execution.</td>
</tr>
<tr>
<td>Correctness</td>
<td>• The degree to which a system or component is free from faults in its specification, design, and implementation.</td>
</tr>
<tr>
<td></td>
<td>• The degree to which software, documentation, or other items meet specified requirements.</td>
</tr>
<tr>
<td></td>
<td>• The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not.</td>
</tr>
<tr>
<td>Verification</td>
<td>• 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.</td>
</tr>
<tr>
<td></td>
<td>• Formal proof of program correctness.</td>
</tr>
<tr>
<td>Validation</td>
<td>The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.</td>
</tr>
</tbody>
</table>
Appendix DevA: CSEG Leads and Component Liaisons

Currently (as of June 2001) CSEG leads and CCSM component liaisons are:

<table>
<thead>
<tr>
<th>Component</th>
<th>Liaison</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSEG Manager</td>
<td>Tony Craig</td>
</tr>
<tr>
<td>CSEG QA Lead</td>
<td>Lawrence Buja</td>
</tr>
<tr>
<td>Component Liaisons</td>
<td></td>
</tr>
<tr>
<td>Atmosphere</td>
<td>Erik Kluzek</td>
</tr>
<tr>
<td>Land</td>
<td>Mariana Vertenstein</td>
</tr>
<tr>
<td>Ocean</td>
<td>Nancy Norton</td>
</tr>
<tr>
<td>Ice</td>
<td>Julie Schramm</td>
</tr>
<tr>
<td>Coupler</td>
<td>Brian Kauffman</td>
</tr>
<tr>
<td>Data Atmosphere</td>
<td>Brian Kauffman</td>
</tr>
<tr>
<td>Data Land</td>
<td>Matthew Hecht</td>
</tr>
<tr>
<td>Data Ocean</td>
<td>Mariana Vertenstein</td>
</tr>
<tr>
<td>Data Ice</td>
<td>Brian Kauffman</td>
</tr>
<tr>
<td>Large Atmosphere</td>
<td>Brian Kauffman</td>
</tr>
<tr>
<td>Shared Utilities</td>
<td>Tony Craig</td>
</tr>
<tr>
<td>Scripts and Build</td>
<td>Lawrence Buja</td>
</tr>
</tbody>
</table>
Appendix DevB: CVS Access Groups

Access to the CCSM repository is controlled through membership in Unix groups and the LockDir feature of CVS-1.11.

The UNIX groups are available only on goldhill (or CGD suns), and are used strictly for access to the repository. (Note: "ccsmr" stands for CCSM repository). In addition the following individuals have access to all groups: tcreag, southern, csm, csm, erik, and mvrtens. These individuals are responsible for tagging the entire system or maintaining CVS access.

<table>
<thead>
<tr>
<th>Group</th>
<th>Description</th>
<th>Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccsmrall</td>
<td>List of everyone that has checkout permission</td>
<td>Everyone that needs checkout access to the CCSM or component models. Also need access to the group &quot;cgdesm&quot;.</td>
</tr>
<tr>
<td>ccsmrclpl</td>
<td>Check-in permission for coupler</td>
<td>Coupler model development team.</td>
</tr>
<tr>
<td>ccsmratm</td>
<td>Check-in permission for active atmospheric models</td>
<td>NCAR and off-site development team for both atmosphere and land-models. Developers working on the &quot;ccm&quot; also need to be in the group &quot;ccsmrln&quot;.</td>
</tr>
<tr>
<td>ccsmrlnl</td>
<td>Check-in permission for active land models</td>
<td>NCAR and off-site development team for both atmosphere and land-models.</td>
</tr>
<tr>
<td>ccsmrocni</td>
<td>Check-in permission for active ocean models</td>
<td>NCAR ocean model development team.</td>
</tr>
<tr>
<td>ccsmrice</td>
<td>Check-in permission for active ice models</td>
<td>Ice model development team both at NCAR and abroad.</td>
</tr>
<tr>
<td>ccsmrdat</td>
<td>Check-in permission for data models</td>
<td>Data model development liaisons</td>
</tr>
<tr>
<td>ccsmruti</td>
<td>Check-in permission for utilities</td>
<td>List of individuals responsible for each utility library</td>
</tr>
<tr>
<td>ccsmrshr</td>
<td>Check-in permission for csm_share</td>
<td>CCSM liaisons</td>
</tr>
<tr>
<td>cgdcvsadmin</td>
<td>Check-in permission to CVS.SCRPIETS</td>
<td>erik,mvrtens</td>
</tr>
</tbody>
</table>

NOTE: What happens if a user tries to commit to a directory they don’t have permission to access? After the log information is entered the commit will fail with a "permission denied" type error as follows:

Checking in nc_fields4.F90;
/fs/cgd/data0/erik/CVS.TEST.REPOS/cpl5/nc_fields4.F90,v <-- nc_fields4.F90
new revision: 1.2; previous revision: 1.1
cvs [commit aborted]: could not open lock file '/fs/cgd/data0/erik/CVS.TEST.REPOS/cpl5/,nc_fields4.F90,'; Permission denied
References


