

# Consortium for Advanced Simulation of LWRs

## CTF Programmer Manual

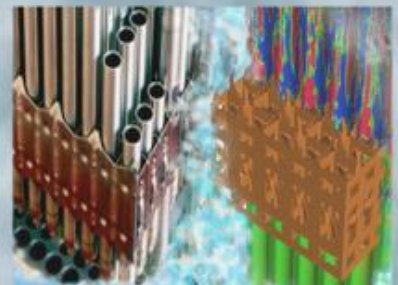
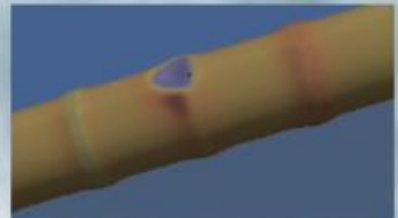
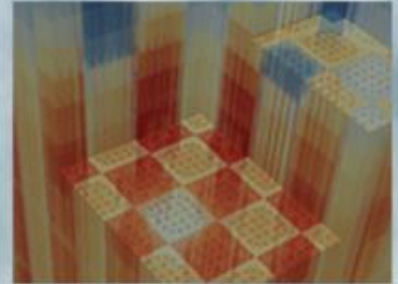
R. Salko<sup>1</sup>

<sup>1</sup>Oak Ridge National Laboratory

11/8/2019



**VERA QA Document** – This document was prepared under the ORNL VERA Quality Assurance Program in accordance with procedure VERA-QA-003. The OFFICIAL COPY of this document is the electronic version in the VERA Documents Repository. Before using a printed copy, verify that it is the most current version by checking the Revision ID against the electronic version.



## Revision Log

---

Revision	Date	Affected Pages	Revision Description
0	3/24/19	All	Initial Release
1	11/8/19	All	Updates for 4.1 release

---

## Document pages that are:

Export Controlled:	None
IP/Proprietary/NDA Controlled:	None
Sensitive Controlled:	None
Unlimited:	All

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

# CTF Programmer Manual

## Approvals:

---

Robert Salko, CTF Product Software Manager

---

Date

---

Aaron Wysocki, Independent Reviewer

---

Date

# Executive Summary

This document provides information needed by developers who will be working in the CTF source code. It provides a high-level overview of code design as well as guidelines for writing source code.

# Contents

Executive Summary . . . . .	iv
Acronyms . . . . .	vi
<b>1 Call Chart</b>	<b>1</b>
<b>2 Style Guide</b>	<b>4</b>
2.1 Introduction . . . . .	4
2.2 High Level Guidelines . . . . .	4
2.3 Documentation . . . . .	6
2.4 Module Design . . . . .	6
2.5 Derived Type Design . . . . .	7
2.6 Procedure Design . . . . .	8
2.7 General Style . . . . .	8
2.8 File Naming . . . . .	8
2.9 Indentation Rules . . . . .	9
<b>Bibliography</b>	<b>10</b>

# List of Figures

1.1	High level call chart for CTF . . . . .	2
1.2	Expansion of the “trans” procedure calling sequence . . . . .	2
2.1	Example of a proper design for a module header section and a derived type (note derived type is partially folded to show entire module header). . . . .	11
2.2	Example of proper procedure declaration, including header documentation and argument declaration . . . . .	12

# Acronyms

# 1. Call Chart

Figure 1.1 shows a very high level call chart for CTF. The call chart uses the actual procedure names starting with the main program name, “cobra\_tf”. A brief description of what the procedure does is provided where needed. More detailed procedure information (including procedure arguments and definitions) can be found directly in the procedure header in the CTF source code or Doxygen documentation.

As the legend indicates, blue boxes denote a procedure that makes calls to other procedures and green boxes denote that the procedure makes no further calls. “cobra\_tf” makes a call to “Init\_tf”, which handles all read-in from the input deck and initialization of data structures in the code. Some of the data initialization needs to be done before the input file is read and some needs to be done after, which accounts for there being two stages of initialization. The “trans” procedure handles solving the transient solution. CTF can be run in pseudo-transient mode, which means the solution will be checked for steadiness during the transient performed in “trans”.

Once the transient ends or the solution is considered steady, code cleanup is performed. Note that this flow denotes what is done during a standalone CTF simulation. When CTF is run in a coupled environment, the solution algorithm is driven by an external code. The process shown in Figure 1.1 is not performed because the main program, “cobra\_tf”, is not called. Rather, the coupled code will call the initialization and cleanup. The solution step can be called multiple times during a coupled simulation.

The “trans” procedure is expanded further in Figure 1.2. The “pre\_trans” procedure is called only once during a standalone simulation; it will be done once per coupled iteration during a coupled simulation. After this, the timestep iteration loop begins. In terms of the CTF solution, this can be thought of as the outer iteration loop. During each iteration, the timestep size is first set using information from the previous timestep (“Timestep”), then boundary conditions are set in “Prep3d” (this must be done each iteration because boundary conditions can be time-dependent), then the solid conduction equations are solved, which includes updating the fluid/solid heat transfer coefficients and solving the temperature distribution in all solid objects in the model, and finally, the fluid governing equations are solved in “Outer\_iteration”.

The name, “Outer\_iteration” is a bit misleading. The governing equation solution is done in one iteration; there is no outer iteration loop. The assumption is made that the timestep size will be small enough that no iteration is required for the governing equations. Note that the governing equations are solved via an operator split method (i.e., solve momentum first, mass/energy second, and transport equations third) as described in the CTF Theory Manual [1].

Once the governing equations are solved, CTF will check to see if the solution is steady (if running a pseudo-transient) or if it has reached the end of the transient (if running a transient). The “post\_trans” step involves



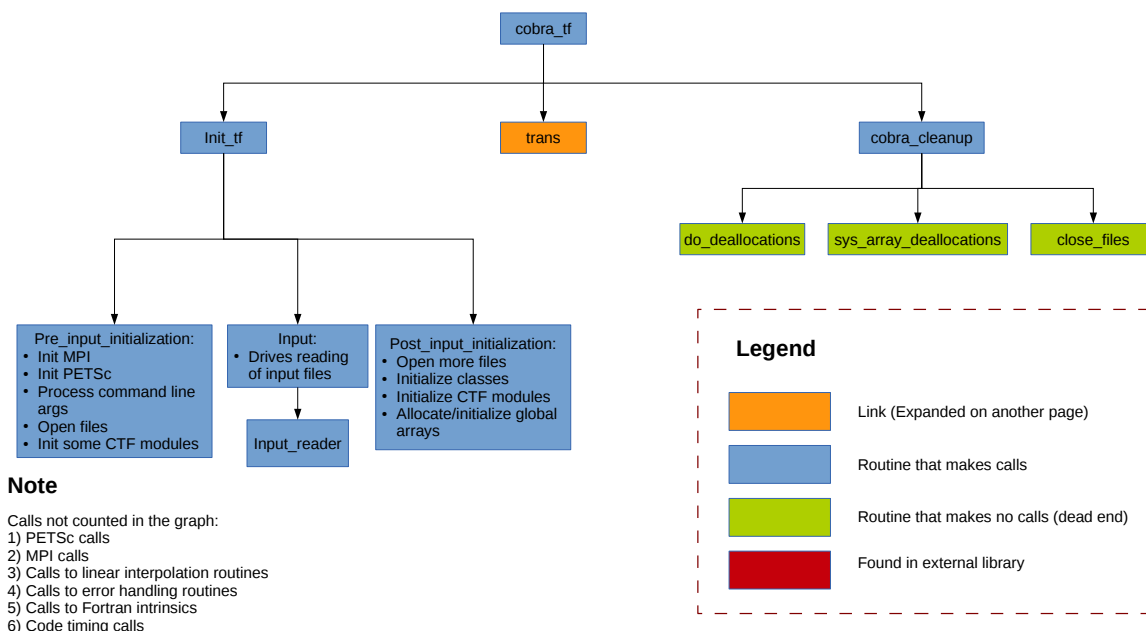


Figure 1.1: High level call chart for CTF

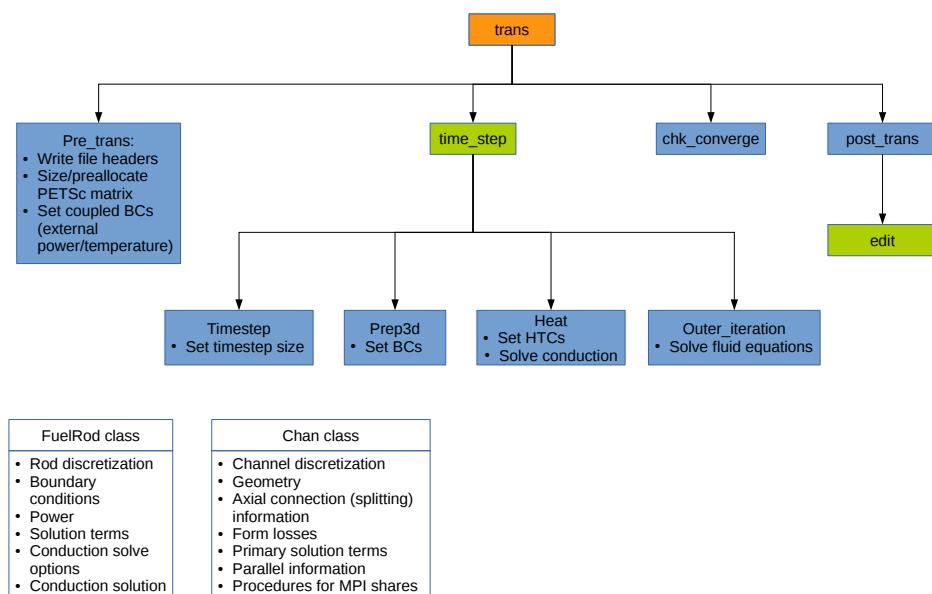


Figure 1.2: Expansion of the “trans” procedure calling sequence

writing output edits. The output edits will only be written here if the code determines it is time to write edits and if the code is running in standalone mode. If coupled, this step is skipped because the coupled driver is in charge of determining when edits shall be written. If the “chk\_converge” step determined the simulation is over, the “trans” procedure will be exited. Otherwise, the iteration loop will continue.

Additional details on the call sequence in CTF can be found in the CTF source code or Doxygen documentation.

Figure 1.2 also shows simplified descriptions of two of the major data structures employed in CTF: the “FuelRod” class and the “Chan” class. The “FuelRod” class contains attributes that define the discretization, solution data, and boundary conditions of a solid heated object in the model as well as the methods necessary to perform solid object solutions and extract information from the object. The class is instantiated once for each solid object (e.g., wall, tube, solid rod, or fuel rod) in the model. Another class called “Conductor” is similar to the “FuelRod”, but applies only to unheated solid objects in the model. In the future, the solid object classes will likely be refactored so that there are separate classes for each solid type modeled by CTF, with all classes being extensions of a single solid base class.

Likewise, the “Chan” class houses all attributes and methods important to the channels in the CTF model. Like the “FuelRod” class, most solution data is stored in 1D arrays that take the axial location index as input. This reflects the pseudo 1D solution nature of CTF.

## 2. Style Guide

### 2.1 Introduction

This document provides instructions for adding to or modifying the CTF source code. These instructions relate to the coding styles that should be employed to ensure a uniform look of the source code. The instructions have been organized into a set categories that span the range of activities a CTF developer would be involved in. An attempt has been made to also touch on general good coding practices; however, it is noted that many volumes have been dedicated to the topic of “good coding practices” and so this set of guidelines only glazes over the high level points. Therefore, each category of instructions has been divided into two sections: (1) a set of **Rules** that are clear, straightforward, and *must* be adhered to by all CTF developers and, (2) a set of **Guidelines** that define ideals that may or may not apply in all situations. We should strive for ideals, but there are instances when it makes sense to violate the ideal in favor of practicality, and so it is left to the subjectivity and creativity of the developer in determining how to adapt to the

### 2.2 High Level Guidelines

#### Rules:

1. Use free-format for writing source.
2. Intrinsic names should be all lowercase.
3. Do not use `goto` statements.
4. Do not use `common` blocks.
5. Do not use the `save` attribute or expect for variables to be saved between procedure calls. CTF is frequently used by a driver code that expects to be able to clear and re-initialize all variables and the `save` attribute does not allow this behavior.
6. Do not use numbered do loops.
7. Do not use Fortran statements and intrinsic function names as variable names (e.g., do not create a variable named `program`, `intent`, `sin`, etc.).

8. The name of a derived type should be written in CamelCase, where each word is capitalized and there are no underscores.
9. Use the exception handler module included in the COBRA-TF repository for raising code exceptions; do not just use `print` and `stop` statements.
10. Use modern logical expressions (e.g., use `==` and `>=` rather than `.eq.` and `.ge.`).
11. The default units in CTF are English units (e.g., BTU, ft, F). The units of any procedure argument or global data must be in English units. If a model that uses non-English units is implemented into CTF, the units should be converted inside the procedure where the model is implemented. All units conversions should be done using the units conversion module in CTF rather than using un-named conversion constants.

### Guidelines:

- Use design-by-contract (DBC) statements provided by Futility to ensure that procedure inputs are valid.
- CamelCase is the preferred way to separate words in derived type, module, and variable names. Module and derived type names should start with a capital letter. Variables should start with a lower case letter. In some cases, a word in a name may be all capital letters, in which case, it may make more sense to use an underscore to separate words for better clarity.
- Do not use un-named constants in your coding (e.g., declare a parameter for PI instead of writing 3.14159). There are very few instances where it is permissible to violate this rule, but it is acceptable to write mathematical expressions out with un-named constants for the sake of clarity. For example, when doubling a number, it is permissible to write `2.0*variable` rather than declaring `2.0` as a `parameter`. Likewise, it is permissible to take the inverse of a variable as `1.0/variable` rather than declaring `1.0` as a `parameter`.
- Use a modular design:
  - Use an object-oriented programming style, making use of derived types and modules whenever appropriate.
  - Encapsulate data and procedures in modules and derived types whenever possible.
  - Design modules to have clear, well-documented interfaces.
  - Many small procedures that each do one thing are better than one giant procedure that does everything.
  - A well-designed class or module is one that is clear and straightforward when creating unit tests on the class or module.
- Use dynamic memory allocation instead of statically-sized arrays whenever possible for code flexibility and efficient memory usage.
- Make variables more self-documenting by using descriptive names (e.g. `bundlePitch` instead of `bp`). Use accepted abbreviations when possible (e.g., “tmp” instead of “temporary”).
- Well designed unit tests should almost always be written for new code additions.

- Add automated regression testing for new features using TriBITS.
- Avoid the use of the `pointer` attribute when `allocatable` can be used in its place.
- Use loop labels when it clarifies loop structures (e.g. multiple nested loops or loops that expand over many pages of code) or when it is necessary for advanced use of `exit` and `cycle` statements.

## 2.3 Documentation

### Rules:

1. Use `Doxygen markup language` to document all modules, derived types, procedures, global data, and procedure arguments.

### Guidelines:

- Good source code should be self-documenting. Use comments generously to describe what is happening, what variables do, and **the units of the variable**.

## 2.4 Module Design

### Rules:

1. What must be included in a module header section (see Figure 2.1 for an example of a proper module header):
  - (a) Description of the module using Doxygen markup.
  - (b) Module dependencies listed using the `use` statement. All module dependencies should be listed in the module header rather than in module procedures. This better clarifies what the module dependencies are.
  - (c) The `implicit none` statement.
  - (d) The `private` statement.
  - (e) The `public` statement followed by all derived types, data, and procedures that are to be visible outside the module. Do not add the `public` attribute to public data to make it public. Creating a single list at the top of the module makes it clear what is public in the module.
2. Do not put more than one module in a single file.
3. Modules should have initialize and clean procedures that are used to initialize and free all module data.

### Guidelines:

- Use modules to organize procedures and data. Functions, procedures, and data that are naturally linked should be grouped in modules. Data and procedures that operate on that data should be grouped into the same module to allow for more efficient encapsulation.
- Multiple derived types should only be grouped in a single module when the types are closely related or when one type extends the other and needs to make use of private data in the base type. Otherwise, a module should house only one derived type.
- Unused variables should be removed from newly developed modules (these can be detected by compiling with the `-Wunused` option in `gcc`).

## 2.5 Derived Type Design

### Rules:

1. A standard derived type header should precede each derived type (see Figure 2.1 for an example).
2. The passed-object in derived type type-bound methods should be named “me”.
3. Every variable in a derived type should be documented using Doxygen markup. If the variable is an array, the meaning of the array indices must be explained. The units of the variable should be included. The variable documentation should be on its own line that directly precedes the variable declaration.
4. A derived type must have a constructor procedure that allocates all arrays used by the object and initializes all data. The constructor shall be a function that returns the instantiated object of the derived type. The function name shall be of the form “init\_ClassName”, where “ClassName” is the name of the derived type.
5. The header section of the type shall include the `private` statement and any data that needs to be `public` should have the `public` attribute added.
6. Any type-bound procedures not needed outside of the type should have the `private` attribute added.
7. The type must include a “clean” procedure that frees all the memory that was used by the type. The procedure must be declared as “final” so that it is automatically called when the type object goes out of scope. If the ability to manually destroy the object is needed, the “clean” procedure can be made non-“final” and a “final” procedure should be created that calls the “clean” procedure.

### Guidelines:

- The derived type should generally have one constructor that completely prepares the object for use. In some cases, the object may need to be built in steps, which requires several setup procedures. In this case, DBC should be used to ensure that type-bound procedures are not used before the object is fully instantiated.
- Mark as much derived type data `private` as possible and maintain a clear, well-documented interface.
- Unit test the derived type to ensure it behaves as intended.

## 2.6 Procedure Design

### Rules:

1. All procedures must be contained in a module.
2. The standard procedure header should precede every procedure (see Figure 2.2 for an example). All arguments shall be documented when the procedure is public.
3. The `intent` of every argument must be marked (e.g. `intent(in)`, `intent(out)`).

### Guidelines:

- Avoid the use of the `intent(in out)`.
- Documenting arguments in private procedures is optional, but it is encouraged when documentation would significantly improve on clarity of procedure purpose and usage. In some cases, argument name is sufficient documentation.

## 2.7 General Style

### Rules:

1. Do not make direct calls to MPI or HDF5. Use the stubs that were created to limit the need for preprocessing statements.

### Guidelines

- Limit number of characters in a line of code to 100 characters. It is permissible to extend beyond this limit at times, but keep in mind that long lines of code require line-wrapping or horizontal scrolling, both of which make source code difficult to read.

## 2.8 File Naming

### Rules:

1. The first letter in the name should be capitalized.
2. If the modules primary purpose is to house a dervied type, its name should be the name of the derived type followed by “\_type”.
3. The module name and the file name it resides in should be identical (e.g. the “TheDerivedType\_type” module should be in the “TheDerivedType\_type.f90” file.)
4. Always use the “.f90” suffix for Fortran source code files.

## 2.9 Indentation Rules

### Rules:

1. An indent is 3 spaces.
2. Do not use the tab character.
3. Indent all `do/enddo` blocks.
4. Indent all `if/elseif/else/endif` blocks.
5. Indent all `type` constructs.
6. Indent all `select case` blocks.
7. Indent all `select type` blocks.
8. Indent code within the `contains` section of modules and derived types.
9. Indent all code within a `subroutine/function`.
10. Indent all continuation lines.



**Guidelines.** This document is part of the CTF Training Plan and must be read by all CTF developers. Note that, being a legacy code, portions of the CTF source do not adhere to this guideline; therefore, it is crucial to be mindful of the instructions in this guide when writing new source or modifying old source rather than simply “doing as has been done before”.

```

!> This module stores the ChanMap derived type.
module ChanMap_type
use AssemMap_type, only: AssemMap
use RodMap_type,   only: RodMap
use ChanGeo_type,  only: ChanGeo
use Preproc_exception_handler, only: msg, Print_error
implicit none
private
public :: ChanMap

!#####
!> Houses a collection of channel maps. Each map is a top-view of the CTF
!! model. A map array takes a channel row and column and returns some data
!! for the channel the row/column points to. All maps are global, meaning the
!! map represents the entire model and not just a single domain.
!#####
type :: ChanMap
!> Takes entity global row and column and returns
!! the unique index of the entity
integer, allocatable :: index_map(:, :)
!> A map that takes global row/col and returns assembly
integer, allocatable :: owner_map(:, :)
!> Gives a value for each channel that says where
!! the channel lies in the symmetry map. The meaning of the values
!! are as follows:
!!   0 means there's nothing there
!!   1 means it's a normal channel
!!   2 means the channel sits on a vertical symmetry line
!!     (this is a half channel)
!!   3 means the channel sits on a horizontal symmetry line
!!     (half channel)
!!   4 means the channel is at the center of the core (only for
!!     even rod bundles and will be a quarter or eighth channel)
!!   5 means the channel is on the diagonal centerline
integer, allocatable :: sym_map(:, :)
!--- 28 lines: > The geometry data for each channel-----
procedure :: Rodpercent
end type Chan_map

contains

```

**Figure 2.1:** Example of a proper design for a module header section and a derived type (note derived type is partially folded to show entire module header).

```

=====
!> Instantiates the ChanMap type
!>
!> @param me The passed dummy object
!> @param gam The instantiated AssemMap object
!> @param sym_opt The symmetry option being employed. See geo_dat GeoInp
!! object doc.
!> @param grm The instantiated RodMap object
!> @param bundlepitch The distance [m] between adjacent bundles
!> @param model_shroud Set to .true. if there is a shroud around the core
!> @param shroud_assem_gap The gap between the assembly boundary and the
!! shroud [m]
!> @param rodpitch The distance between adjacent rod centers [m]
!> @param nrodsrow The number of rods in a row of an assembly
!> @param wallflag Set to 1 if there is a wall around the assembly
=====
subroutine Initialize(me,gam,sym_opt,grm,bundlepitch,model_shroud,&
                    shroud_assem_gap,rodpitch,nrodsrow,wallflag)
    class(ChanMap), intent(in out) :: me
    type(AssemMap), intent(in) :: gam
    type(RodMap), intent(in) :: grm
    real, intent(in) :: bundlepitch, rodpitch
    integer, intent(in) :: sym_opt, nrodsrow, wallflag
    logical, intent(in) :: model_shroud
    real, intent(in) :: shroud_assem_gap
    --- 22 lines: integer :: totchansrow, totchanscol-----
end subroutine Initialize

```

**Figure 2.2:** Example of proper procedure declaration, including header documentation and argument declaration

# Bibliography

- [1] R.K. Salko and M.N. Avramova. *CTF Theory Manual*. The Pennsylvania State University.