INL/EXT-13-29228

RELAP5-3D Developer Guidelines and Programming Practices

George L. Mesina

May 2013

INL/EXT-13-29228

RELAP5-3D Developer Guidelines and Programming Practices

George L. Mesina

May 2013

Idaho National Laboratory
Thermal Science and Safety Analysis
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
Office of Naval Reactors
and for the
U.S. Department of Energy
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

INL/EXT-13-29228
Revision 0

RELAP5-3D Developer Guidelines and Programming Practices

Dr. George L Mesina

May 16, 2013

Implementation of a New DTSTEP Algorithm for use in RELAP5-3D and PVMEXEC Completion Report

Dr. George L Mesina

May, 2013

Idaho National Laboratory
Thermal Science and Safety Analysis
Idaho Falls, Idaho 83415
http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Naval Reactors
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

RELAP5-3D Developer Guidelines and Programming Practices

Implementation of a New DTSTEP Algorithm for use in RELAP5-3D and PVMEXEC Completion Report

INL/EXT-13-29228

May, 2013

Approved by:

Dr. George Mesina Author	Date
Nolan Anderson Technical Reviewer	Date
Dr. James Wolf Project Manager	Date
George Griffith Department Manager	Date

EXECUTIVE SUMMARY

Our ultimate goal is to create and maintain RELAP5-3D as the best software tool available to analyze nuclear power plants. This begins with writing excellent programming and requires thorough testing. This document covers development of RELAP5-3D software.

These guidelines are intended to institutionalize a consistent way of writing programming for the RELAP5-3D computer program that will minimize errors and rework. A common format and organization of program units creates a unifying look and feel to the code. This in turn increases readability and reduces time required for maintenance, development and debugging. It also aids new programmers in reading and understanding the program. Therefore, when undertaking development of the RELAP5-3D computer program, the programmer must write computer code that follows these guidelines.

This set of programming guidelines creates a framework of good programming practices, such as initialization, structured programming, and vector-friendly coding. It sets out formatting rules for lines of code, such as indentation, capitalization, spacing, etc. It creates limits on program units, such as subprograms, functions, and modules. It establishes documentation guidance on internal comments.

The guidelines apply to both existing and new subprograms. They are written for both FORTRAN 77 and FORTRAN 95. The guidelines are not so rigorous as to inhibit a programmer’s unique style, but do restrict the variations in acceptable coding to create sufficient commonality that new readers will find the coding in each new subroutine familiar.

It is recognized that this is a “living” document and must be updated as languages, compilers, and computer hardware and software evolve.

CONTENTS

1.0 Purpose and Scope 6
2.0 RELAP5-3D Source Code 7
2.1 Library Quality Programming 7
2.1.1 Major Sections 7
2.1.2 Structured Programming Paradigm 7
2.1.3 Vector Loop Processing 8
2.1.4 Parallel Code 8
2.2 Formatting Statements of Source Code 9
1.1.1 Spacing 9
2.2.5 Case of Letters in Statements 9
2.2.6 Indentation 10
2.2.7 Continuation Line 10
2.2.8 Error Message Formats 10
2.2.9 Warning Message Formats 10
2.3 Programming Rules 10
2.3.1 FORTRAN Statements 11
2.3.2 Unacceptable FORTRAN Statements 11
2.3.3 FORTRAN Subprogram Comments 12
2.3.4 Pre-compiler Directives 12
2.4 Program Units 12
2.4.1 Subprograms 12
2.4.2 Modules 13
3.0 Living Document 14

1.0 Purpose and Scope

Our ultimate goal is to create and maintain the best software tool available to analyze nuclear power plants. This is a lofty ideal. It begins with having excellent programming and requires thorough testing. In this section, the primary goals for RELAP5-3D software development are stated. These goals are for writing style, computability, and code speed. Code accuracy and robustness are not addressed in these goals.

2.0 RELAP5-3D Source Code

2.1 Library Quality Programming

RELAP5-3D program units should have the professional look of other library software such as IMSL or those published in books like Numerical Recipes. Moreover, subprograms should be written to produce the correct result while running as quickly as possible. These organization of program units is explained in subsections 2.1.1 and 2.1.2 while code run speed is addressed in subsections 2.1.3 and 2.1.4.

2.1.1 Major Sections

Modules, subroutines, and functions have major sections that occur in a particular order to create uniformity and ease in locating information.

For a FORTRAN 95 module the major sections are
(1) Module header statement
(2) Module identification comments after module statement
(3) PART 1 – Declaration section
(4) PART 2 – Documentation section
(5) PART 3 – Internal subprograms

The major elements of a subprogram unit (whether independent or internal) occur in the following order:
(1) Subprogram header statement
(2) Subprogram identification comments after subprogram statement
(3) Declaration section organized according to Section 2.2
(4) Data dictionary that defines all local variables and call arguments
(5) An "Executable Code" comment that separates the non-executable top of the program unit from the executable section
(6) Outline style comments throughout the body of the executable code
(7) Body of the program unit written in uniform programming style in accordance with the rest of this guideline

2.1.2 Structured Programming Paradigm

Structured programming is considered the most strongly modular form of coding, even stronger than object-oriented programming. It makes a code very readable.

A structured program unit is written as a sequence of blocks-of-code and sub-blocks-of-code. A block-of-code in FORTRAN is one of the following programming constructs: do-loop, if-then-else, case, and non-branching statement (assignment, I/O, etc.).

The following rules are strictly enforced to have structured programming:
(1) No jumps (via go to statements) into a block or sub-block from outside it
(2) No jumps from inside a block or sub-block to outside it
    (a) Jumps may only terminate inside the block or at its bottom end
(3) No backwards jumps except via a do-loop construct.

(4) Blocks may not co-terminate. Thus for example, an outer and inner do-loop may not end on the same statement.

2.1.3 Vector Loop Processing

Modern computer chips are designed with short vector loops that provide excellent code runtime reduction for loops that vectorize. Therefore, loops should be written to allow vector speed-up where possible. This is especially important in transient coding which is rerun with each call of its encompassing subprogram unit and may be executed hundreds, thousands, even millions of times.

Vector loops should have no recursion, subprogram calls, I/O, or inner loops. Even loops with an inner loop may vectorize on come platforms if the inner loop satisfies certain conditions. Also, calls to subprograms will not inhibit vectorization if they are intrinsic functions, statement functions, or can be internalized to the body of the loop through a compiler directive or option.

To develop vector loops, it is necessary to characterize them. The simplest characterization of a vector loop that has no calls or I/O and satisfies the following:

> *If the original loop were replaced by a collection of small loops, each having exactly one of its statements and the same do-loop index, start, end, and skip-factor, then the same final values would be calculated*.

This is typically how vector calculations are performed.

2.1.4 Parallel Code

Parallelism in RELAP5-3D is implemented with openMP paradigm. This paradigm applies to shared memory multi-core computer chips, which are found on most workstation and personal computer today.

Use of the openMP paradigm requires placement of openMP directives in the code to break out parallel sections and parallel loops. It also requires the use of a compiler option, generally given through a command line option, to activate it.

Shared memory parallelism is somewhat less restrictive than vectorization, but is nonetheless easy to break with recursion. It can also result in different calculations when the same loop is run several times due to the order of operations being done differently each time. Moreover it is very important not to parallelize any loop of section of code that is not inherently parallel, or the computer will generate incorrect results.

Preprocessors can detect parallelism inhibitors, but are not perfect and can indicate that a perfectly good loop should not be parallelized. It is safest to not force parallelization through use of directives that override the pre-compiler analysis.

(1) All Vector loops are parallel loops.
(2) I/O does not inhibit parallel.
(3) Calls to subprograms may or may not inhibit parallelism.
(4) Inner loops do not inhibit parallelism.
(5) Recursion inhibits parallelism.

(6) Parallelism can lead to different results on each run due to floating point round-off.

2.2 Formatting Statements of Source Code

The following is a set of programming requirements and conventions for formatting lines of code. This is intended to make the code more readable and easier to maintain. There are two reasons for doing this. The first is to establish a uniform coding style that is flexible enough to allow some individuality, yet allow everyone familiar with the style to recognize other people's programming quickly and easily. The second is to allow new developers with a programming background in C++, Java or similar languages to quickly become productive with RELAP5-3D by employing modern programming constructs and styles with which they've become familiar.

2.2.1 Spacing

1) Each FORTRAN keyword should have a space on either side of it. Keywords that have an argument list should have a space after the closing parenthesis.
2) The elements of a keyword's argument should have a space after each comma, though this is optional.
3) Lists of variables should have a single space after each comma. On I/O statements the space after comma is optional though preferred.
4) Assignment and do statement *equal signs* should have a space on either side.
5) Spaces should be placed on either side of + and - arithmetic operator. Exception: omit spaces around + and – inside index calculations.
6) Do not put spaces around **, * and / operators.
7) Spaces around logical operators are optional.

For example, replace

```
do   i1=1,nvar,2
  i=i1+j1*ncolumns+k1*nplanes
  read(unit,format,end=100,err=200)a,b,c(i+1)
```

by

```
do i1 = 1, nvar, 2
  i1 = i1 + j1*ncolumns + k1*nplanes
  read (unit, format, end=100, err=200) a, b, c(i+1)
```

2.2.5 Case of Letters in Statements

Upper and lower case letters can be used to aid readability. Just like in a book, most text should be in lower case. Use of upper case can improve readability if used in a predictable and uniform manner.

All FORTRAN 77 executable source code should be in lower case. This does not apply to comments.

FORTRAN 95 source should either use lower case or camelback notation for variable names. Camelback puts a capital at the start of each word within a variable name. For example:

- auxfilename is lower case
- auxFileName is camelback.

FORTRAN 95 keywords may use all capital letters; this is optional.

2.2.6 Indentation

In some RELAP5-3D program units, indentation can be very deep, exceeding 10 levels and even reaching 15 in places. To prevent long and deeply indented FORTRAN 77 statements having too many continuation lines, the standard of two space indentation was created. For uniformity, this is applied to FORTRAN 95 coding also.

Initial indentation and length of lines in a statement are also made uniform. The following rules apply:

1) Indentation is two spaces for each level of indentation.
2) Do not use tabs for indentation in FORTRAN coding.
3) Leftmost coding in FORTRAN 77 coding is column 8.
4) Leftmost coding in FORTRAN 95 goes in column 2.
5) Comments begin in column 1 in both FORTRAN 77 an 95 coding; however, FORTRAN 95 coding indent comments if it improves readability.

2.2.7 Continuation Line

In FORTRAN 77 statements, use the “&” ampersand symbol in column 6.

In FORTRAN 95 statements, use the “&” ampersand symbol at the end of the line at least 10 spaces to the right of the last non-blank unless it is arranged in the same column as an ampersand in the line immediately above or below it. Lining up continuation marks improves readability.

2.2.8 Error Message Formats

Formats that write an error message should begin with ‘0********’ (in other words, a zero followed by eight asterisks). This is important to those who teach RELAP5 training courses.

2.2.9 Warning Message Formats

Formats that write a warning message should begin with ‘0$$$$$$$$’ (in other words, a zero followed by eight dollar signs). This is also important to those who teach RELAP training classes.

2.3 Programming Rules

The following is a set of programming requirements and conventions. Much of this is intended to make the code more readable and easier to maintain. There is need to eliminate certain archaic programming practices that are hard to read, time-consuming to decipher, prone to difficulty debugging, and/or have been listed as obsolete or deprecated in the FORTRAN 95 ANSI standard.

2.3.1 FORTRAN Statements

It is allowable to use every type of non-obsolesced statement in the ANSI FORTRAN 77 and FORTRAN 95 standard in the files or the respective types.

1. Use “implicit none” in every program unit, module and subprogram.
2. Use the module named intrtype in all program units.
3. Declare every variable in a program unit.
4. Alphabetize declaration lists for each data type.
5. The order of declaration statements is:
    a. Derived type creation, derived type instances, arrays, scalars
    b. Declare FORTRAN basic types in this order:
        i. Integer
        ii. Real
        iii. Logical
        iv. Character
6. Initialize all variables.
7. Nullify all pointers
    a. When they are created, if that does not ruin the algorithm.
8. Deallocate all subtypes of derived type quantities before deallocating the instance of a derived type.
    a. This eliminates memory leaks.
9. Use format statements unless the format can fit on one line within the confines of the argument list of the I/O statement.

2.3.2 Unacceptable FORTRAN Statements

Many features have become obsolescent in later FORTRAN standards. Though these statements are still legal FORTRAN, they should not be used in any new coding. Moreover, there are many perfectly legitimate FORTRAN constructs and statements that should also not be used in any new coding. These items have caused much difficulty with debugging and maintenance and give new programmers great trouble.

1. Do not introduce any NEW *bit-packing*.
2. Do not introduce any new *assigned go to* or *computed go to* statements.
3. Do not use *backward go to* statements. See the code architect if you cannot find a way to avoid introducing one.
4. Replacement: FORTRAN 90 “case” statement or FORTRAN 77 multiple elseif conditional statement (if/then/else/else if/ … /else if/end if).
5. Do not introduce new *equivalence or common* statements.
6. Never introduce machine-dependent compiler extensions. Stick with the ANSI FORTRAN 95 standard.
7. Do not exceed array index bounds. Checking array bounds with compiler options is encouraged.
8. *Avoid* placing *multiple return* statements in a program unit.

2.3.3 FORTRAN Subprogram Comments

Internal documentation is important to identify the program unit, author, creation dates as well as give its purpose, provide a dictionary of its variables, and outline its operations. This section formalizes the kinds and locations of comments in program units. It is reasonable to have between 25% and 50% of the lines of code be comments.

1. Subroutine identification documentation includes
    a. subprogram purpose
    b. creation/update date
    c. cognizant engineer
2. Data Dictionary should occur after the declarations
    a. Alphabetize the dictionary variables
    b. Use Capital letters in the definition to show how the variable name relates.
    c. Line up the definitions to all start in the same column.
3. Comments within the body of the subprogram should be outline style.
    a. Each major section should receive a section title and short description.
    b. Each important minor section should receive a subsection title.
    c. Each heading should be numbered outline style (publication style)
        Example: a major section heading might be "1.0 Input."
        Example: a minor section heading might be "1.1 Argument checking."
4. Blank Comment Line
    a. A blank comment may precede a significant comment line for readability.
5. Spacing
    a. Each comment should have a ! in column 1 and the text should begin in column 4.
    b. Begin each sentence/phrase with a capital letter and end with a period.

2.3.4 Pre-compiler Directives

1) Do not introduce any pre-compiler directives that have a line count. For example:

```
#if def,mystuff,1
        Write (*,*) diagnostic
```

This should be coded as:

```
#ifdef mystuff
        Write (*,*) diagnostic
#endif
```

2) Do not begin pre-compiler directives names with a number.
3) Do not name pre-compiler directives with a common word as those words can be removed throughout the code by the pre-compiler.

2.4 Program Units

2.4.1 Subprograms

Subprograms should not have so many lines of coding that they become hard to read, understand, debug, develop, and maintain.

New subprogram should not exceed 200 lines. If one becomes too long in its main routine, break out subsections to create internal subprograms or new external subprograms.

Internal subprograms should not exceed 200 lines, and with rare exceptions, should be shorter than the main subprogram to which they are internal. Internal subprograms can be further divided into smaller internal subroutines, and one internal routine can call another.

2.4.2 Modules

Use statements should make use of the "only" construct. Limit the items "used" in a module or subprogram to just those required. For example, if variables AA and BB are the only items from module ABC, then rather than

```
        use abc
```

program

```
        use abc, only: aa, bb
```

Beware of using modules inside other modules. The only exception is intrtype which should be used in all RELAP5-3D modules.

Using one module inside another creates a dependency and forces order precedence on compilation. This means the used module must be compiled before its user-module. This can create recursion (E.G. Module A uses Module B which uses Module C which uses Module A) and makes compilation impossible until it is resolved.

In general, it is best combine modules in subprograms external to modules to prevent compilation order requirements.

Module internal subroutines should primarily work on data declared within the module itself. If it requires data from another module, that data should come through the argument list. If so much data is required from other sources that the call list becomes unwieldy, the subprogram should be promoted to an external subprogram.

2.5 Executable Programming Rules

Here are some rules for acceptable and unacceptable programming practices.

NO allocate/deallocate statements in the transient.
This has at least two negative effects on code performance. First, the unnecessary finding and setting aside of memory causes slowdowns. Second, it prevents shared-memory parallelism unless coupled by appropriate protection, namely, that only the first thread is allowed to allocate the memory. A subsequent attempt by another thread to allocate the same memory is an error.

Deallocate all allocated memory ASAP
RELAP5-3D can have multiple cases in a single input deck. Failure to deallocate can cause memory leaks and core dumps.

Initialize/Nullify pointers at creation
As soon as a pointer is created it should be initialized. Unless it is pointed to an existing value immediately upon creation, it should be nullified immediately.

Initialize variables at creation
Many global variables are created by modules. Most existing modules provide an “init” subroutine for initializing some or all of their variables. New modules should provide similar subroutines. Subroutine modmem also calls “init” subroutines from some modules.

In the case of other global variables, subroutine initdata provides the appropriate platform for initializing data.

Local variables should be initialized at the beginning of the subprogram. If the values must be saved from one call to the next, the initialization section should be protected by an if-test on a logical variable that is set to true before the subprogram is called and false inside the if-block. Note that this variable should be initialized in subroutine initdata so that it can be reset to true for each new case of an input deck.

3.0 Living Document

It is recognized that this is a “living” document and must be updated as languages, compilers, and computer hardware and software evolve. If you have questions, comments, or suggestions for improvements, please contact Dr. George Mesina at the Idaho National Laboratory. Your input will be evaluated and possibly included in future editions of the RELAP5-3D Developer Guidelines.