Developer Guidelines for RELAP5-3D Programming, 2013

Dr. George Mesina

RELAP5 International Users Seminar
Idaho Falls, ID
September 12, 2013

Outline

- Programming Goals
- Source Code
- Code Behavior
- Testing
- Documents

Programming Goals

- Major Goals: SQA, Debugging/Maintenance, Speed
- Quality
  - Bug prevention – unit testing
  - Verification – indicate how coding implements model in report/ document
  - Validation – supply test cases to validate new coding
    - Against analytical results
    - Against appropriate data if available
    - Against other computer codes if possible

Programming Goals

- Structured programming
  - MUCH easier to debug.
  - Program units comprised of a series of coding blocks where each block has EXACTLY one entry at top and one exit at bottom.
  - Blocks may have sub blocks.
  - Structured programming has stronger modularity than OOP.

Programming Goals

- Run speed
  - Vector/parallel coding generally runs faster, even on serial/scalar machines.
  - Vector programming – loops without certain features
    - Recursion, I/O, sub-loops, calls to non-inline subprograms
    - PCs now have short vectors to speed execution
  - SMD Parallel programming – loops without data dependency
    - Recursion, thread order issues
    - Many multi-core machines allow SMD

Source Code

- Program Units
- Source Coding
- Source Code Formatting

Program Units

- Main Program
- Module
- Subprogram
  - Subroutine
  - Function
  - Intrinsic
  - Blockdata

Module

- Name ends in “mod” and should be 9 letters or less.
- Internal form – 3 sections
  1. Declarations
     - Avoid USE statements (except level 0 modules)
  2. Data Dictionary
  3. Internal Subprograms
- 1. Declarations – 4 subsections
  - Derived type definitions
  - Derived types
  - Arrays
  - Scalars
  - *Alphabetize* the variable names of each basic type

Modules

- Data Dictionary
  - Derived types first, *alphabetical* listing of variables, regardless of basic type
  - Remaining data in *alphabetical* order, regardless of basic type
- Module subprograms
  - Restrict to work on the *module's data*. EXAMPLES:
    - Constructor, destructor, restart writer, restart reader, calculations with module data
  - If any external data is needed, bring it in through call parameters.
    - No USE statements
    - If call sequence gets too long, remove subprogram from module.

Subprograms

- Main subprograms vs. internal subprograms
  - Main subprogram has the contains statement
- Main – description, declarations, dictionary, body, internal routines
  - Description: documentation of purpose, author, date
  - Declarations: Same order/alphabetization rule as modules
  - Data Dictionary: Same as modules
- Body of main subprogram
  - Outline style comments precede each major structured programming block of coding.
  - Outline major sub-blocks. Explain important points too.
  - Long sections of coding, particularly pre-compiler protected code, can be made into internal subprograms
  - NO restriction on USE statements in MAIN subprograms

Subprograms

- Internal subprograms
  - Description required
  - Author and date optional (normally not needed)
  - Declarations, dictionary (or local variables), and body rules the same as for main subroutine
  - Place needed USE statements into containing program unit:
    - Main subprogram or Module declarations
    - Helps various debugging/maintenance efforts

Source Code Programming

- Employ ANSI Standard FORTRAN *only*
  - No compiler extensions such as real do loop indices
- No obsolescent Fortran or any of the following:
  - Equivalence, common, bit-packing, backward go-to, etc.
- Use error trapping on read, write, open, allocate, deallocate statements
- Memory leak prevention
  - Test *before* allocating and deallocating
  - Deallocate from bottom up
- Initialization: Nullify all pointers and initialize all variables ASAP
- No allocate or deallocate in transient (except reflood).

Source Code Format

- F90+ continuation mark >= 5 spaces after last non-blank
- Lower case except in comments and *camelBack* variable names.
- Spaces *around* =/::/+/-/comparator signs and *after* keywords and commas (except inside array references)
- Indentation: 0 spaces for continuation lines, 2 for sub-blocks.
- Precompiler directives: OpenMP, Vector, and CPP/FPP/GPP only
  - !$omp, !cdir$, #ifdef, #ifndef, #endif, #else, #include
- Use same documentation as for modules and additionally:
  - Subprograms place “Executable Code” comment before first such line
  - Document important/tricky points for the next guy, he may be you!

Code Behavior

- Goals
  - If possible, process all input, using defaults to replace user errors, and give user good messages.
  - Code should detect inability to proceed, write a message, and stop on its own; not abort with a core dump or hang the machine.
- Messages (input and elsewhere):
  - Error Messages start with “0********”
    - Identify the source (input card, fluid property, file, etc.) as specifically as possible (Word on card, quantity, filename, etc.)
  - Warning Messages start with “0$$$$$$$$”
    - Ignored input, replacement cards, replacement values, etc.
  - Informational Messages have no special start
    - Input edits, output edits, status of transient, etc.

Code Behavior

- File Operations
  - Do not overwrite special files: input, property, restart, printed-output (the last one has a special command line override)
  - Issue error message (screen and output file) if user:
    - Attempts to overwrite special file
    - Required input is not found
    - Set failure flag for graceful shutdown
  - Do not open or close files in the transient
    - Slows code and breaks parallel
- Input
  - A new card requires new (internal) subroutine, messages, & edit.
  - For errors, provide messages and, if possible, default values
  - If required input cannot be defaulted, give an error message and terminate immediately by calling “abort.”

Code Behavior

- Input Cases
  - Be careful that new data is deallocated at the end of a case, and at start of next case, re-allocated and re-initialized
  - If there is an error in a previous case that set the fail flag, don't run.
- Run Termination
  - Immediate failure – set fail flag, write message, call abort.
    - Used if proceeding would cause a core dump. E.G.
      - File unavailable, out of memory/time, machine hang, singular coefficient matrix, variable has impossible value
  - Graceful failure – set fail flag, write error message, proceed to end of section (input or transient) where diagnostics are printed.
    - Allows final dump on output files.
  - Normal Termination – final writes, deallocate memory, close files

Code Behavior: Code Output

- Printed output file, outdta
  - Add new output to appropriate section (volume data in volume output block of major edit, minor edits in minor edit area, etc.)
  - For significantly different data, create it in an appropriate spot
    - E.G. Coriolis Effect would go in TH area
  - Coding goes in MAJOUT or IMIEDT
- Restart
  - Add new data to the read and write subroutines of the appropriate module(s).
- New files
  - Ensure naming (command line, input card, default), file open and close, output control (from DTSTEP)

Testing

- For new subroutine, develop a unit testing program to call and test it
- Develop one or more test cases that test it from within RELAP5-3D
- Run installation test suite:
  - Make sure it affects no other calculations, unless it is supposed to
  - If it should affect calculations (bug fix, model improvement), justify that it does so correctly.
- INL runs additional test suites when code updates are added.
  - Developmental Assessment
  - Verification Test Suite
  - DTSTEP Test Matrix
  - Others

Documentation

- See RIUS 2011 “RELAP5-3D Architecture and Style” for details.
- See G Mesina, “RELAP5-3D Developer Guidelines and Programming Practices,” Revision 1, INL/EXT-13-29228, June 2013.
- It will become part of Vol. 8 of the RELAP5-3D manuals when that is produced.