VERA Virtual Environment for Reactor Applications

ORNL/SPR-2021/2327

MPACT Programmer's Manual Version 4.3

June 23, 2022

Aaron Graham[1], Dan Jabaay[2], Yuxuan Liu[2], Brendan Kochunas[2], Shane Stimpson[3], and Ben Collins[4]

[1]Oak Ridge National Laboratory
[2]University of Michigan
[3]BWXT
[4]University of Texas

OAK RIDGE National Laboratory

U.S. DEPARTMENT OF ENERGY

ORNL is managed by UT-Battelle LLC for the US Department of Energy.

ORNL/SPR-2021/2327

MPACT PROGRAMMER'S MANUAL VERSION 4.3

Aaron Graham[1], Dan Jabaay[2], Yuxuan Liu[2], Brendan Kochunas[2], Shane Stimpson[3], and Ben Collins[4]

[1]Oak Ridge National Laboratory
[2]University of Michigan
[3]BWXT
[4]University of Texas

Date Published: June 23, 2022

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

MPACT Programmer's Manual Version 4.3

Revision Log

Revision	Date	Affected Pages	Revision Description
0	06/14/2022	All	Updated document to VERA template and generated new document number; this supersedes previous document version CASL-U-2019-1876-000

Document pages that are:

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

MPACT Programmer's Manual Version 4.3

Approvals:

Aaron Graham, MPACT Product Software Manager	**Date**
Erik Walker, Independent Reviewer	**Date**

CONTENTS

LIST OF FIGURES

1. CALL CHART

1.1 CODE EXECUTION

Figure 1 shows a high-level call chart for MPACT. The call chart uses the actual procedure names, starting with the main program name, **MPACT**. The only exception is the italicized names where the upstream procedure contains a number of subroutines calls. These calls are logically separated into blocks, such as *setup basic info/inputs*. Brief descriptions of what the procedure accomplishes are provided where needed. More detailed procedure information, including procedure arguments and definitions, can be found directly in the procedure header in the MPACT source code or Doxygen documentation (http://www.doxygen.nl/).

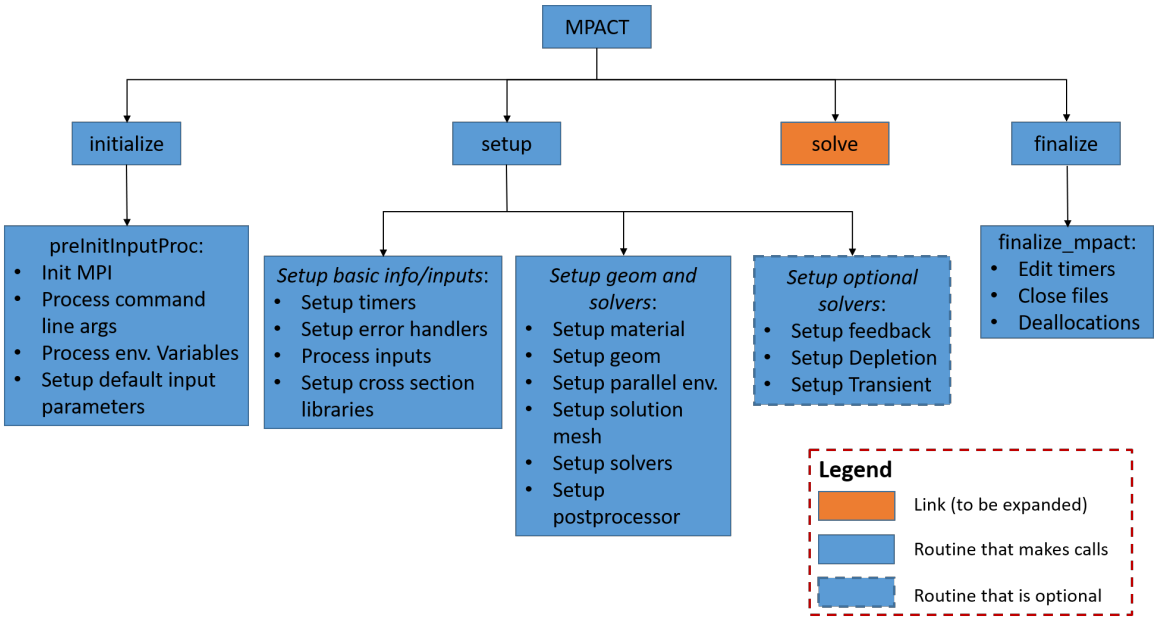

Figure 1. High-level call chart for MPACT.

MPACT first makes a call to **initialize**, which processes the command line arguments and sets up the default input parameters. The read-in from the input deck and the initialization of data structures are performed at the beginning of setup, and then all the geometry and meshes are initialized. As the figure legend indicates, the dashed blue boxes are optional routes, depending on the user options. For example, if thermal feedback is enabled in the input, then the feedback solver will be initialized in the **setup** procedure. The single orange **solve** box represents the procedure actually solving the problem, which is expanded in Fig. 2. Once the solution converges, the code is finalized, as represented by the **finalize** box.

1.2 SOLVE EXECUTION

The **solve** procedure is expanded further in Figure 2. For nominal core depletion, the multistate solver drives the downstream solvers to perform depletion and run iterations between eigenvalue solve and thermal hydraulic (TH) feedback solve. Once the solution is converged for the current state, the postprocessor is called to write output edits, and then the calculation moves forward to the next state. As noted in the figure, the call sequences of transient calculation differ from those of steady-state calculation. However, the steady-state calculation is always performed for the initial state. The transient multilevel (TML) solver is the default option which allows large transient time steps.

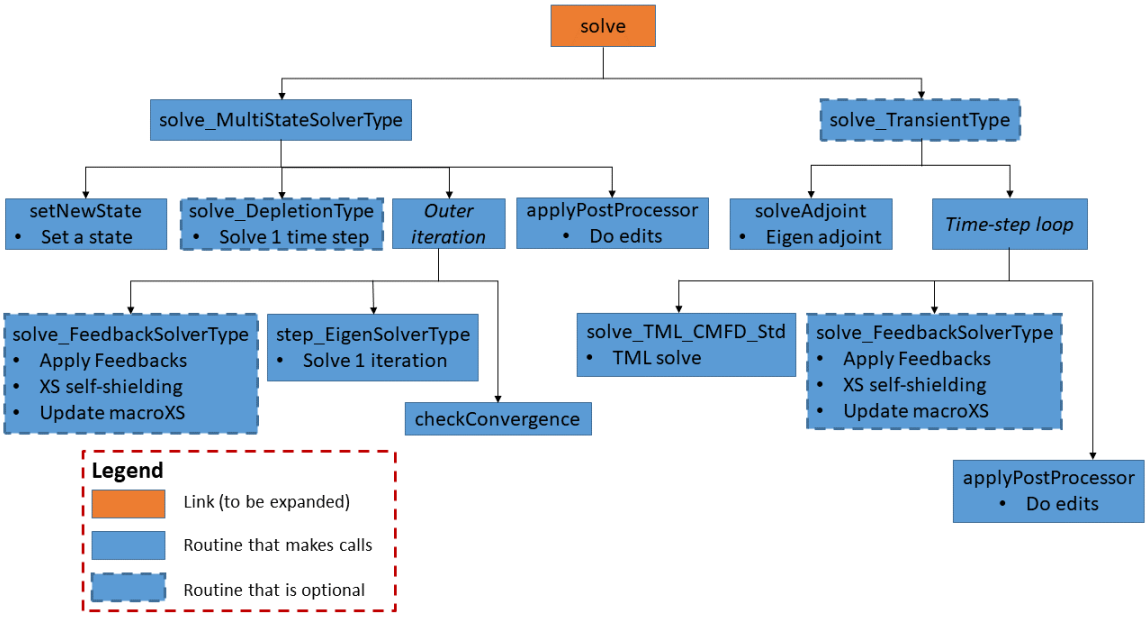

Figure 2. Expansion of the "solve" procedure calling sequence, which progresses from left to right.

2. STYLE GUIDE

2.1 INTRODUCTION

2.1.1 Scope and Objectives

This document outlines the requirements for writing code in MPACT, Futility, and MAMBA. Instructions address coding style, to include writing understandable, maintainable source code, as well as guidelines on how the source code should look.

2.1.2 How to Read this Document

This document is intended to function as a reference for developers. It identifies *rules* and *guidelines* for formatting Fortran source code. Style guides and standards for all other languages are described in Section 2.1.3. Key terms are defined as follows:

- **rule**: a principle that must always followed or implemented by every developer.
- **guideline**: a principle that is recommended as a best practice and should be implemented whenever possible.
- **reason**: rationale for requiring guidelines or rules, sometimes based on the developer's judgement.

Where applicable, the reason for a rule or guideline is given. Each section of this guide is intended to be read individually as needed, and not sequentially from start to finish. Therefore, rules and guidelines are cross referenced between the various sections to guide the reader to relevant information about related rules and guidelines.

2.1.3 Non-Fortran Language Standards and Style Guides

The remaining sections of this document include rules and guidelines for Fortran source code. This sub-subsection defines which style guides and standards should be used for other programming languages and domain languages found in the MPACT and Futility repositories.

- C style guide
  For C code, the low-level virtual machine (LLVM) clang format shall be used. Formatting may be automated with the clang format tool:
  https://llvm.org/docs/CodingStandards.html (accessed 12/18/2018)
- C++ style guide
  For C++ code, the LLVM clang format shall be used. Formatting may be automated with the clang format tool.
  https://llvm.org/docs/CodingStandards.html (accessed 12/18/2018)
- Python style guide
  For all Python code, the PEP 8 Style guide shall be used.
  https://www.python.org/dev/peps/pep-0008/ (accessed 12/18/2018)
- Bash and Shell style guide
  For Bash and other Linux/Unix shell languages, the style guide from Google shall be used.
  https://google.github.io/styleguide/shell.xml (Rev. 1.26, accessed 12/18/2018)
- CMake style guide
  For CMake related files, the style guide used by the KDE project shall be used.
  https://community.kde.org/Policies/CMake_Coding_Style (accessed 12/18/2018)

- YAML style guide - No style guide.
- Perl style guide - No style guide.
- LaTeX style guide - No style guide.
- All other languages
  This is a category for any other languages that may not be explicitly defined above. Some examples of these languages include Docker files, XML, and Doxygen configuration files.

Any language or file format that does not have an explicit style guide has no official rules or guidelines regarding style.

2.2 GENERAL STANDARDS

2.2.1 Rules

1. Use free-formatted Fortran for writing source.
   REASON: compilers no longer require strict column formatting, and adhering to this is overly burdensome to the developers.
2. All source shall be written to the Fortran 2003 standard [1].
   REASON: This is easily enforceable by the compiler, and it is formal standard.
3. Use all upper case letters for all Fortran Keywords and Intrinsic functions.
   REASON: Allows for clear comprehension of Fortran keywords and intrinsics.
4. Do not use the following prohibited keywords.
   - `GOTO`
     REASON: This execution construct is too general, and it greatly reduces comprehension of program flow.
   - `EQUIVALENCE`
     REASON: Comprehension of a program is easier when each variable has its own dedicated memory.
   - `DATA`
     REASON: It is better to define variable values with initialization statements.
   - `ENTRY`
     REASON: This execution construct obfuscates program flow and testing. Separate procedures are to be used instead.
5. Use the following assumed variable units.
   - Density is g/cc
   - Number density is particles/barn-cm
   - Temperature is Kelvin
   - Pressure is MPa
   - Flow rate is kg/s
   - Power is MW
   - Length is cm
   - Area is cm^2
   - Volume is cm^3
6. Use modern relational operators.
   REASON: Operators are similar to other languages, so readability and comprehension is improved for non-Fortran programmers.

Old operator (Do not use)	New operator (Use)	Meaning
`.EQ.`	==	Equal to
`.NE.`	/=	Not equal to
`.GT.`	>	Greater than
`.LT.`	<	Less than
`.GE.`	>=	Greater than or equal to
`.LE.`	<=	Less than or equal to

7. Do not include implicitly defined variables.
   REASON: Implicitly defined variables complicate debugging and can lead to unintended consequences.

8. Do not allow defined entities (e.g., variables, modules, derived types, procedures) to share the name of a Fortran keyword.
   REASON: This prevents ambiguous code.

9. When the `END` keyword appears, do not include a single space between `END` and the name of the construct being ended.

Incorrect	Correct
`END`	`END <construct_name>`
`END IF`	`ENDIF`
`END DO`	`ENDDO <loop label, if there is one>`
`END SUBROUTINE <subroutine_name>`	`ENDSUBROUTINE <subroutine_name>`
`END FUNCTION <subroutine_name>`	`ENDFUNCTION <subroutine_name>`

   REASON: The optional second word prevents ambiguous code.

10. All Fortran source files shall have the projectâĂŹs appropriate header at the top.

11. Keyword phrases for which the language requires two or more keywords shall be written with a single space between each keyword in the phrase.
    EXAMPLE: `SELECT CASE`, `SELECT TYPE`, `TYPE IS`, `CLASS DEFAULT`, `END INTERFACE`, etc.
    REASON: Compilers and editors will always support the keywords separated by a space, but they will not always support combined keywords.

12. The following text shall appear before a `CONTAINS` statement:

```
!
!
!===============================================================================
```

    REASON: Helps denote the file structure.

2.2.2 Guidelines

1. All procedures should make use of the defined `REQUIRES` and `ENSURES` macros provided by Futility for Design-by-Contract.

2. Semicolons can be used to improve readability in some cases, such as when introducing a long series of `IF` or `CASE` statements with only one line of executable code; use whitespace appropriately with the semicolons to further improve readability.

3. Encapsulate literal strings using single quotes, not double quotes.
   REASON: String literals typically must include a literal double quote in the string.

2.3 MODULE DESIGN

2.3.1 Rules

1. All modules shall contain a header with the format shown below. The appropriate header is from RULE 2.2.1.10, or:

```
!> @brief A brief description of the purpose of the module
!>
!> A detailed description of the purpose of this module and description of
any
!> public classes or procedures defined here. Include a list of any
procedures
!> provided for testing and list high level requirements implemented by the
!> module
!
!++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++!

MODULE <name>
#include "Futility_DBC.h"
USE Futility_DBC

IMPLICIT NONE
PRIVATE
```

   REASON: Provides consistent look, feel, and documentation.

2. Do not use the `ONLY` statement with module `USE` statements.
   REASON: This is difficult and onerous to maintain.

3. Always include `IMPLICIT NONE` after `USE` statements.
   REASON: Supports RULE 2.2.1.7.

4. Other interfaces and module global variables defined in a `MODULE` before the `CONTAINS` statement shall include Doxygen comments describing their purpose.
   REASON: Provides consistent look, feel, and documentation of members.
   EXAMPLE:

```
!> String for module name
CHARACTER(LEN=*),PARAMETER :: modName="ExampleModule"

INTERFACE Example_RoutineAB
!> @copybrief ExampleModule::RoutineA
!> @copydetails ExampleModule::RoutineA
MODULE PROCEDURE RoutineA
!> @copybrief ExampleModule::RoutineB
!> @copydetails ExampleModule::RoutineB
MODULE PROCEDURE RoutineB
END INTERFACE Example_RoutineAB

!> Number of instances of the type that have been created
INTEGER(SIK),SAVE :: nInstances=0
```

5. Any procedures that must be accessible outside the module shall be explicitly declared `PUBLIC`, with one entity per statement.
   EXAMPLE: `PUBLIC :: Mod_PublicFunction`
   REASON: This practice promotes better encapsulation.

6. Module names shall start with a capital letter and use CamelCase for the module name, and they shall include the suffix Module.
   REASON: Derived types also start with capital letters and have CamelCase names. The suffix Module clarifies what the entity is. The suffix is added to the module because modules are referenced less frequently than derived types.
7. There shall be one module per file for source code. Unit tests are an exception.
   REASON: This facilitates organizing the code structure and navigating source.
8. Filenames shall be the same as the module they contain without the Module suffix.
   REASON: This facilitates organizing the code structure and navigating source.
   EXAMPLE: File FutilityComputingEnvironment.f90 defines the module `FutilityComputingEnvironmentModule`.

2.3.2 Guidelines

1. The unit test for a module should be implemented as a `SUBMODULE` which extends the module.
   - This approach allows direct testing of private components and methods of derived types without making them public.
   - A procedure whose interface is defined in the main module and made `PUBLIC` inside of an `#ifdef UNIT_TEST` preprocessor block should be implemented in the `SUBMODULE`. The name of the procedure should be the same as that of the module prepended with "runTest".
   - In the same file as the testing `SUBMODULE`, a `PROGRAM` block should be included which executes the "runTestModule" procedure.
2. Any private procedures or variables that are not part of a derived type should be made available through a wrapper.
   (a) The wrapper should be prefixed with "test_".
   (b) The wrapping procedure should have the same interface as the procedure it wraps.
   (c) The wrapper should be explicitly declared `PUBLIC` inside of an `#ifdef UNIT_TEST` preprocessor block.
   (d) The better way of doing this would be do use the previously described "SUBMODULE" testing approach and "IMPORT" the non-`PUBLIC` objects. However, the current versions of *gfortran officially supported by VERA do not implement this capability even though it is definied in the Fortran Standard.*

2.4 DERIVED TYPE DESIGN

2.4.1 Rules

1. Any derived type components shall have Doxygen comments describing their purpose.
   NOTE: Doxygen's website is here: http://www.doxygen.nl/
2. Derived types shall be declared with default `PRIVATE` components.
3. Derived type type-bound procedure definitions shall be indented under the `CONTAINS` statement.
4. Only one component shall be defined per line.
5. Derived type component declaration attributes shall be declared on the same line.
6. All derived type components shall have a defined initial value.
   NOTE: that allocatable and derived type components cannot have explicitly defined initial values in their declaration. Allocatables are by default unallocated, which is a defined state, and derived type initial values are determined in the type declaration.
   NOTE: Pointer attributes shall be initialized to null, as in
   `INTEGER(SIK),POINTER :: somePtr(:) => NULL()`.
7. All type-bound procedures shall be explicitly declared as `PASS` or `NOPASS`.
8. The following rules shall be followed when extending a derived type and overriding procedures:
   8.1. Requirements for the overriding procedure shall never be more strict than those for the overridden procedure.
   8.2. Requirements for the overriding procedure shall never be less strict than those for the overridden procedure.

2.4.2 Guidelines

1. Constructor methods should be defined with the name **init**.
2. Destructor methods should be defined with the name **clear**.
3. Define a component as `LOGICAL(SBK) :: isinit=.FALSE.` to indicate the initialization status for the derived type.
4. Extensions to base types should include the base type name with the suffix "_<Ext>", where <Ext> represents the extension. This is the only exception to the CamelCase rule that should generally be followed when naming.
   NOTE: For deep class hierarchies, the intermediate types do not need to be included as the suffix. The suffix should be chosen to maximize clarity.
5. Type-bound procedures should include comments in the following form:

   ```
   !> @copybrief   <Module Name>::<Routine Name>
   !> @copydetails <Module Name>::<Routine Name>
   ```

   For DEFERRED type-bound procedures, the same type of comments should be used, with <Routine Name> referring to the `ABSTRACT INTERFACE`, if one is used to define the `DEFERRED` procedure interface.
6. Type-bound generics do not require Doxygen comments.
7. For each component's comment, include useful information regarding the use of the component, expected values, units, and the shape/size for allocatable entities.
8. Use accessors on derived types as much as possible to prevent client code from directly modifying components of a type.

2.4.3 Example

An example derived type definition is provided below:

```
!> @brief  Brief description of the type
!>
!> Further details describing the derived type.
TYPE :: DerivedType
PRIVATE
!> Initialization status
LOGICAL(SBK) :: isinit=.FALSE.
!> Description of component1
INTEGER(SIK),PUBLIC :: component1=0
!> Description of component2
REAL(SRK),POINTER :: component2(:,:) => NULL()
CONTAINS
!> @copybrief   <Module Name>::Method1
!> @copydetails <Module Name>::Method1
PROCEDURE,NOPASS :: Method1
!> @copybrief   <Module Name>::Method2_DerivedType
!> @copydetails <Module Name>::Method2_DerivedType
PROCEDURE,PASS :: Method2 => Method2_DerivedType
ENDTYPE DerivedType
```

2.5 PROCEDURE DESIGN

2.5.1 Rules

1. Each procedure (function or subroutine) shall have a header that includes descriptions of the procedure and input/output parameters, as well as documentation of the procedures contract as illustrated in the example in Section 2.5.3.
2. All Doxygen comments in procedure headers shall include *requires*, *guarantees*, and *implementation* notes sections.
3. The `INTENT` attribute shall be declared explicitly for all procedure dummy arguments.
4. Each dummy argument to a procedure shall be declared on one line by itself.
5. The return argument of any `FUNCTION` shall be specified with the `RESULT` clause.

2.5.2 Guidelines

1. Use no more than 6 dummy arguments.
   REASON: The need for more than 6 dummy arguments indicates that the routine may be too complex.
2. Avoid the use of the RETURN statement.
   REASON: For clarity of program flow, every routine should strive for one entry point and one exit point.
   EXCEPTION: In some cases the use of a `RETURN` at the beginning of a routine based on a logical condition can help code clarity, particularly for very long routines in which the alternative is the use of a large IF block.
3. Avoid the use of `PURE` and `ELEMENTAL` as procedure attributes when not necessary.
   REASON: Use of these attributes complicates debugging, prevents the use of DBC statements, and does not necessarily guarantee better optimizations.
4. DBC should be used instead of the exception handler when the feedback is intended for a developer and not a user. The exception handler should be used when feedback is intended for the user.
   Generally, DBC items should never have bad values under expected execution of the code. Arguments going to the exception handler are usually more closely tied to a user input.
5. If a procedure has an optional argument requiring large `IF/ELSE` blocks, or if it has multiple optional arguments, then consider splitting into multiple procedures.
   Define an `INTERFACE` or make a method `GENERIC` so the client code can call the routines the same way as before.
6. When dynamic casting to an extended type, avoid large indentation blocks. Instead, assign the extended type to a local pointer.
   EXAMPLE:

```
TYPE(ExtendedType),POINTER :: myExtType

myExtType => NULL()
SELECT TYPE(myClass); TYPE IS(ExtendedType)
myExtType => myClass
ENDSELECT
```

2.5.3 Example

An example procedure header might look like this:

```
!> @brief Brief description of the procedure
!> @param param1 description of first input parameter, if the description is
!>               long, carryover the indentation like this.
!> @param param2 description of parameter 2.
!> @returns val if the procedure is a function the argument to RESULT() should
!>          be commented with the doxygen @returns keyword.
!>
!> Detailed description of the purpose and behavior of the procedure. This
!> description should include the following sections, which include the "contract"
!> of this procedure to its clients.
!>
!> @par Requires:
!>   - List of expected pre-conditions for the input parameters
!> @par Guarantees:
!>   - List of guaranteed outcomes, assuming all requirements were met.
!> @par Implementation Notes:
!>   - Any notes that might be relevant to future developers, such as
!>     why a particular algorithm or data structure was chosen over others
!>     possible optimizations, why a certain dummy argument is required, etc.
!>
```

2.6 UNIT TEST DESIGN

2.6.1 Rules

Use the Futility unit testing framework.

2.6.2 Guidelines

1. The order of the test is recommended as follows:
   - Test the constructor and destructor.
     - The destructor should work, regardless of the object state.
   - Test each type-bound procedure.
   - Test each public procedure.
2. Associate one `SUBTEST` (test subroutine) with one type-bound procedure or procedure.
3. Have a separate `COMPONENT_TEST` for each test case of a method.
4. Include a `COMPONENT_TEST` for the routineâĂŹs behavior in the uninitialized state.
   NOTE: It is not necessary to test DBC checked conditions.
5. Include component tests for each exception that may be raised.
6. Utilize subroutine/sub test local test objects.
7. Have at least one unit test program per module.
8. Use an explicit `ASSERT` for everything that should be checked from a `COMPONENT_TEST`; do not use indirect `ASSERT`s on indirect conditions.
9. Use `ASSERT_FAIL` for checking allocation/association status of arrays and pointers.
   REASON: Subsequent `ASSERTS` typically will access these members, and these will segfault if they are not allocated or associated.
10. When possible, use the derivative `ASSERT_*` statements that automatically provide output upon failure.
    NOTE: The list of assert statements is given in the table below:

Macro	Statement	Description
`ASSERT_EQ`	`a == b`	
`ASSERT_LT`	`a < b`	
`ASSERT_LE`	`a <= b`	
`ASSERT_GT`	`a > b`	
`ASSERT_GE`	`a >= b`	
`ASSERT_APPROXEQ`	`a .APPROXEQ. b`	For approximate equivalence of reals
`ASSERT_APPROXLE`	`a .APPROXLE. b`	Approximately less than or equal
`ASSERT_APPROXGE`	`a .APPROXGE. b`	Approximately greater than or equal
`ASSERT_APPROXEQF`	`a .APPROXEQF. b`	bitwise equivalence of reals
`ASSERT_APPROXEQR`	`a .APPROXEQR. b`	Relative comparison for approximately equal
`ASSERT_APPROXEQA`	`a .APPROXEQA. b`	Absolute comparison for approximately equal
`ASSERT_SOFTEQ`	`SOFTEQ(a,b,c)`	Approximately equal for a specific tolerance

11. When `ASSERT` (or derivatives) appears in a loop, use a `FINFO` statement to provide loop indexes.

2.6.3 Example

Use the appropriate header from RULE 2.2.1.10, or:

```
PROGRAM testSomeModule
#include "UnitTest.h"
USE UnitTest
USE SomeModule

IMPLICIT NONE

CREATE\_TEST("SOME MODULE")

REGISTER_SUBTEST("%clear()",testClear)
REGISTER_SUBTEST("%init(...)",testInit)
REGISTER_SUBTEST("%method1(...)",testMethod1)
REGISTER_SUBTEST("Public_Proc",testPublic_Proc)

FINALIZE_TEST()
!
!===============================================================================
CONTAINS

! ... test routines ...

END PROGRAM testSomeModule
```

2.7 VARIABLE DECLARATION

2.7.1 Rules

1. Do not initialize variables upon declaration unless they have the `PARAMETER` or `SAVE` attribute.
   REASON: Assignment with declaration implies the SAVE attribute. This is unclear and can lead to unintended behavior.

2. Explicitly define the kind of all variables.
   NOTE: The following table lists the individual kinds and their uses:

Kind parameter	Usage	Data type description	Mnemonic
`SBK`	`LOGICAL(SBK)`	32-bit boolean	Selected Boolean Kind
`SNK`	`INTEGER(SNK)`	32-bit integer	Selected Normal Kind
`SLK`	`INTEGER(SLK)`	64-bit integer	Selected Long Kind
`SIK`	`INTEGER(SIK)`	Default integer kind (32-bit)	Selected Integer Kind
`SSK`	`REAL(SSK)`	32-bit real	Selected Single Kind
`SDK`	`REAL(SDK)`	64-bit real	Selected Double Kind
`SRK`	`REAL(SRK)`	Default real kind (64-bit)	Selected Real Kind

3. Each variable attribute shall be listed on the same line as its type.

See also:

- 2.2.1.7 There shall be no implicitly defined variables.
- 2.2.1.8 Defined entities (variables, modules, derived types, procedures, and so on) shall not share the name of a Fortran keyword.
- 2.4.1.4 Only one component shall be defined on a line.
- 2.4.1.6 Each derived type component shall have a defined initial value.
- 2.5.1.3 The `INTENT` attribute shall be declared explicitly for all procedure dummy arguments.
- 2.5.1.4 Each dummy argument to a procedure shall be declared on one line by itself.

2.7.2 Guidelines

1. Have one purpose for each variable.
2. Avoid using the `DIMENSION` attribute; instead, define array variables as var(:).
3. Avoid the use of fixed sized arrays. Use assumed size or variable size arrays or dynamically sized arrays.
   NOTE: Some exceptions may exist for the `CHARACTER` type.
   EXCEPTION: Given specific mathematical formulations, it may be advantageous to use the fixed size array. For example, spatial coordinates in two or three dimensions should use fixed size arrays

2.8 NAMING CONVENTIONS

2.8.1 Rules

1. Variables with the `PARAMETER` attribute shall be named in ALL_CAPS with underscores separating words.
   REASON: This is a common convention in many languages which helps clarify that the variableâĂŹs value cannot change.
2. Modules shall be named with CamelCase, starting with a capital letter and ending with the word *Module*.
3. Derived types shall be named with CamelCase, starting with a capital letter.
4. Type-bound procedures shall use CamelCase but shall start with a lower case letter.
5. Preprocessor symbols defined in the project shall start with "<PROJECT>_".
   NOTE: <PROJECT> would be one of MPACT, FUTILITY, or MAMBA.
6. Local variables shall be camelCase, with the first character always being lowercase.
7. Public procedures shall have the module name (without the "Module" suffix) as a prefix, followed by an underscore and the procedure name.

2.8.2 Guidelines

1. Avoid using type as a suffix for derived types.
   REASON: The usage of a derived type's name is almost always accompanied by the `TYPE` keyword.
2. For extensions of derived types, include a suffix of "_<Ext>", where <Ext> is some meaningful description of the extension.
3. Procedures shared in a generic interface should have a suffix "_<Ext>", where <Ext> is some meaningful description of the variation.
4. The prefixes "my" and "this" should not be used for derived type components.

2.9 COMMENTS

2.9.1 Rules

1. Do not leave large blocks of code that have been commented out.
2. All modules shall include Doxygen documentation.
   See also 2.3.1.1.
3. All derived types shall include Doxygen documentation.
   See also 2.4.1.1 and 2.4.3.
4. All procedures shall include Doxygen documentation.
   See also 2.5.1.1 and 2.5.3.
5. Do not use Doxygen documentation within a procedure.
6. Avoid useless comments.
   - A simple restatement of what the code does is a useless comment.

2.9.2 Guidelines

1. For a set of nested constructs, include comments on the same line as the associated END statement when it enhances clarity.
   NOTE: In these cases, consider refactoring into a procedure, especially if the code is duplicated.
   NOTE: For `DO` loops, the suggested comment is the loop variable name.
2. Use explanatory comments for describing complex code or large blocks of code.
3. When a comment is describing a code's intent, focus on the problem and not the solution.

2.10 WHITESPACE

2.10.1 Rules

1. Do not use tabs; only spaces.
2. Indentation is 2 spaces.
3. All line endings shall be UNIX line endings.
4. Use whitespace around both sides of the pointer assignment operator.
5. Use whitespace around both sides of relational and conditional operators.
6. The first line of every file shall always start in the first column.
7. Include a space after Doxygen comments "!>".
8. Increase the indentation level for the following constructs/keywords.
   8.1 `SUBROUTINE`
   8.2 `FUNCTION`
   8.3 `TYPE`
   8.4 `INTERFACE` / `ABSTRACT INTERFACE`
   8.5 `IF` / `ELSE IF` / `ELSE`
   8.6 `DO` / `DO WHILE` / `FORALL` / `WHERE`
   8.7 `ASSOCIATE`
   8.8 `TYPE IS` / `CLASS IS` / `CLASS DEFAULT`
   8.9 `CASE` / `CASE DEFAULT`
9. Do not indent for `SELECT TYPE`; all key words listed in 2.10.1.8.8 shall align with the `SELECT TYPE`.
10. Do not indent for `SELECT CASE`; all key words listed in 2.10.1.8.9 shall align with the `SELECT CASE`.
11. Indent at least two levels (4 spaces) on line continuation.
12. For line continuation across more than 2 lines, indent all continuing lines to the same level.
13. Never end files with a blank line.
    REASON: Git considers this a whitespace error.
14. Do not have trailing whitespace.
    REASON: Git considers this a whitespace error.

2.10.2 Guidelines

1. Keep all text to within 100 columns, but preferably 80 columns. If the line is longer than 80 columns, then its purpose should be to improve clarity.
   REASON: This fits the width of standard 8.5 × 11 in. paper in Courier 10 pt font, and it fits a half-screen on the majority of laptops. However, sometimes breaking near 80 columns can make line breaks awkward and reduce code clarity.
2. For line continuation, break lines before binary operators.
3. Whitespace may be added for alignment of multiple statements across multiple lines when it improves readability of the code.

REFERENCES

[1] Information technology – programming languages – fortran – part 1: Base language. Technical Report ISO/IEC 1439-1:2004, International Organization for Standardization, Geneva, Switzerland, 2004.