IFE/KR/E-2009/005

PROGRAMMING STANDARDS FOR IFEFEM AND DERIVED APPLICATIONS

Institute for Energy Technology

Institute for Energy Technology

	KJELLER	HALDEN	
Address	NO-2027 Kjeller, Norway	NO-1751 Halden, Norway	Open
Telephone	+47 63 80 60 00	+47 69 21 22 00	
Telefax	+47 63 81 63 56	+47 69 21 22 01	

Report number IFE/KR/E-2009/005	Date 2009-11-16
Report title and subtitle Programming standards for IfeFEM and derived applications Fortran 90/95 version	Number of pages 74
Project/Contract no. and name Modelling of Solar Grade Silicon Crystallization Processes/ 186949/I30	ISSN 0333-2039
Client/Sponsor Organisation and reference Norges forskningsråd	ISBN 978-82-7017-797-4 (printed) 978-82-7017-798-1 (electronic)

Abstract

This document defines a standard for the FORTRAN 90/95 implementation of IfeFEM. The main topics in the standard are the naming convention for all FORTRAN 90/95 entities and the coding style and syntax restrictions. We also introduce templates for the different FORTRAN 90/95 program units. We also briefly discuss a system for automatic reference type documentation of FORTRAN 90/95 implementations. Finally we comment on some tools available to simplify the development of FORTRAN 90/95 code.

It is recommended that the standard defined in this document should be used with all applications, in Ife control, derived from IfeFEM.

Keywords: Programming standard, Fortran 90/95, IfeFEM

	Name	Date	Signature
Author(s)	Magne Rudshaug	19/3 - 2010	M. Rudshaug
Reviewed by	Hallvard G. Fjær	19/3 2010	[illegible]
Approved by	Arild Ek	19/3-10	Arild Ek

Contents

Introduction

The materials modelling group at Institute For Energy Technology has been using Fortran 77 exclusively as their programming language since its arrival. The advent of Fortran 90/95 allows a change towards a modern programming style, but at the same time preserves our investment in Fortran 77 code. This quality alone makes Fortran 90/95 the obvious choice compared to competing programming languages. A considerable gain will also be realized by the fact that it is much simpler to learn the new features of Fortran 90/95 than to make a complete transition to another programming language like C++. Fortran 90/95 can offer data hiding and an object based programming style, but not a truly object oriented programming style.

When we introduce a new programming language we have a unique opportunity to define a local programming standard. But what should the concern of a local programming standard for Fortran 90/95 be? In this document we take the view that a local standard for Fortran 90/95 should be concerned with the following issues:

- Restrict the alternative syntax possibilities.
- Make recommendations with respect to the use of new features.
- Definition of a naming convention for Fortran 90/95 entities.
- Documentation of Fortran 90/95 program units.

The main purpose of this document is to define a programming and programmers interface documentation standard for the IfeFEM 3.0 project coded in the Fortran 90/95 language. It is our ambition that the programming style used in the Fortran 90/95 version of IfeFEM 3.0 will be based on the conventions, recommendations and naming conventions defined in this document. The IfeFEM 3.0 project will benefit in a number of ways from the introduction of a programming and documentation standard.

- A unified programming style improves the legibility of the code
- An automated and unified style of documentation will make it easier to keep the documentation updated.
- The introduction of a new documentation format, HTML, will simplify maintenance and reuse of code.
- A programming standard simplifies the introduction of new people into the project.
- The quality of the code will improve and thus the reliability of IfeFEM 3.0 as a tool for the development of high quality finite element code.

In the present form this document should be considered as a working document and will be subject to revisions. We start the standard document by defining a general naming convention for the Fortran 90/95 entity names.

Naming conventions

A source code is simpler to read when a well defined naming convention is used. The main qualities of a naming convention are to make the named entities in the program recognizable and their meaning more easily understood. By recognizable we mean:

- Recognizing the scope of the entity - local or global
- Recognizing the type of the entity - program, module, procedure, assignment operator subroutine, operator function, operator or variable (intrinsic, derived type).
- Recognizing from what application the entity originates.

To accommodate this we introduce a general definition of a Fortran 90/95 entity name.

Definition 1 *A Fortran 90/95 entity name shall have the general form:*

```
[prefix]word1[sep][word2][sep][word3]
```

The `prefix` shall consist of letters and possibly underscore, `_`, characters. The `prefix` is used to make the entity name recognizable. As separator, `sep`, we shall use the underscore character. `word1`, `word2` and `word3` are the components of the entity name and shall consist of letters and numerals. The first character of a component shall be a letter. The components shall define the meaning of the entity name.

As an example of the use of this definition we will make a name for a subroutine that calculates the row sum of an matrix. One suggestion would be:

```
subRow_Sum
```

Here we identify `sub` as the `prefix`, the underscore as the separator and `Row` and `Sum` as the components `word1` and `word2` respectively. We refer to the entity name as a *construction* if it consists of more than one component. The part of the entity name consisting of the components and separators only is referred to as the *basename* . The basename is formed by disregarding the `prefix`. In the example above the basename is `Row_Sum`.

We now use this definition to give a naming convention for the different entities in the Fortran 90/95 programming language.

Application and library names

To easily recognize where a Fortran 90/95 entity name originates we define a unique prefix. To simplify the construction of a unique prefix we introduce the *application acronym*. The term application acronym will also be associated with a library. For the application acronym we introduce the convention:

Naming convention 1 *`Application acronym:` An application or library shall have a unique acronym of no more than two letters based on the name of the application or library. The acronym shall consist of small letters only.*

Particularly for the IfeFEM 3.0 library we introduce the convention:

Naming convention 2 *The IfeFEM 3.0 library shall have the application acronym `f`. Derived applications shall use a different acronym.*

In the examples below we will use IfeFEM 3.0 's application acronym.

Program names

The name of a program should be easily distinguishable from other entities. We have therefore introduced the following naming convention for it.

Naming convention 3 *`Program:` Shall have a prefix consisting of `p` appended with the application acronym. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : pfProgramName

Module names

The name of a module should be easily distinguishable from other entities. We have therefore introduced the following naming convention for it.

Naming convention 4 *`Module:` Shall have a prefix consisting of `m` appended with the application acronym. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : mfModuleName

To accommodate the construction of unique module data and procedure names we introduce the the *module acronym* .

Naming convention 5 *`Module acronym:` A module name shall have a unique acronym of no more than six letters based on the module name components. The acronym shall consist of capital letters and numerals only.*

In the examples below we will use the module acronym `GRID`.

Procedure and operator names

Procedures discussed here are all module procedures. There is no inherent difference between module procedures. However, we find it useful to make the following distinction:

- Subroutine's defining assignment operators. We will refer to them as *assignment operator subroutines*. The assignment operator subroutines shall be private. The assignment operator, (=), shall be public if the associated derived data type is public.
- Function's defining operators. We will refer to them as *operator functions*. The operator functions shall be private. The name of the associated operator shall be public if the associated derived data type is public.
- Procedures that are not assignment operator subroutines or operator functions. These will again be split in three.
    - An internal procedure with respect to a module procedure will be referred to as a *local procedure*.
    - A private or local procedure with respect to a module will be referred to as a *module local procedure*.
    - A public or global procedure with respect to a module will be referred to as a *module global procedure*.

We now introduce a naming convention for each type of procedure as defined above.

Naming convention 6 *`Assignment operator subroutine`: Shall have the prefix `a`. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : aAssignmentSubroutineName

Naming convention 7 *`Operator function`: Shall have the prefix `fo` appended with a character identifying the result data type of the function, `i` for integer type, `r` for real type, `z` for complex type, `l` for logical type, `c` for character type and `t` for a derived data type. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : fotOperatorFunctionName

Naming convention 8 *`Local procedure`: A subroutine shall have no prefix. A function shall have a prefix consisting of `f` appended with a character identifying the result data type of the function, `i` for integer type, `r` for real type, `z` for complex type, `l` for logical type, `c` for character type and `t` for a derived data type and an underscore character. The components shall consist of lower case letters and numerals. Separators shall be used in constructions.*

Example : subroutine_name, fr_function_name

Naming convention 9 *`Module local procedure`: A subroutine shall have no prefix. A function shall have a prefix consisting of `f` appended with a character identifying the result data type of the function, `i` for integer type, `r` for real type, `z` for complex type, `l` for logical type, `c` for character type and `t` for a derived data type. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : SubroutineName, frFunctionName

Naming convention 10 *`Module global procedure:` A subroutine shall have a prefix consisting of the module acronym appended with an underscore character and the application acronym. A function shall have a prefix consisting of the module acronym appended with an underscore character, `f`, a character identifying the result data type of the function, `i` for integer type, `r` for real type, `z` for complex type, `l` for logical type, `c` for character type and `t` for a derived data type and the application acronym. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : MODEX_fSubroutineName, MODEX_frfFunctionName

Closely associated with the operator function is the name of the operator (user defined operator). Operators may be categorized according to their scope as follows:

- Local with respect to the module. The scope of the operator is the module. We will refer to this operator as a *local operator*.
- Global with respect to the module. The scope of the operator is potentially all modules. We will refer to this operator as a *global operator*.

We now introduce a naming convention for each type of operator defined above:

Naming convention 11 *`Local operator name:` Shall have the prefix `o`. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : oOperatorName

Naming convention 12 *`Global operator name:` Shall have a prefix consisting of the module acronym appended with an underscore character, `o`, and the application acronym. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : MODEX_ofOperatorName

Derived data type definition names

Derived data type definitions may be categorized according to their scope as follows:

- Local with respect to the module. The scope of the derived data type is the module. We will refer to this derived type as *local derived data type definition*.
- Global with respect to the module. The scope of the derived data type is potentially all modules. We will refer to this derived type as *global derived data type definition*.

We introduce a naming convention for each of the derived data type definitions defined above.

Naming convention 13 *`Local derived data type definition:` Shall have the prefix `d`. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : ***dDerivedDataTypeDefinitionName***

Naming convention 14 *`Global derived data type definition:` Shall have a prefix consisting of the module acronym appended with an underscore character, the letter `d` and the application acronym. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : ***MODEX_dfDerivedDataTypeDefinitionName***

In the definition of derived data types we use intrinsic and derived data types as components. We will refer to these as the *derived data type components*. Their scope is simply the derived type definition. We introduce the following naming convention:

Naming convention 15 *`Derived data type component:` Shall have a one character prefix identifying the data type of the variable, `i` for integer type, `r` for real type, `z` for complex type, `l` for logical type, `c` for character type and `t` for a derived data type. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : ***zDerivedDataTypeComponentName***

Named constants

Named constants or parameters may be categorized according to their scope as follows:

- Local with respect to a procedure. The scope of this named constant is the procedure and possibly local procedures. We will refer to this type of named constant as a *procedure local parameter*.
- Local with respect to a local procedure. The scope of this named constant is the local procedure. We will refer to this type of named constant as a *local procedure local parameter*.
- Local with respect to a module. The scope of this named constant is the module or the procedure. We will refer to this type of named constant as a *local parameter*.
- Global with respect to a module. The scope of this named constant is potentially all modules. We refer to this type of named constant as a *global parameter*.

We now introduce a naming convention for each type of named constants defined above.

Naming convention 16 *`Procedure local parameter:` Shall have a one character prefix identifying the data type of the parameter, `i` for integer type, `r` for real type, `z` for complex type, `l` for logical type, `c` for character type and `t` for a derived data type* *appended with the underscore character. The components shall consist of capitalized letters and numerals. The components shall be separated by a separator character.*

Example : ***l_PARAMETER_NAME***

Naming convention 17 `Local procedure local parameter:` *Shall have a one character prefix identifying the data type of the parameter,* `i` *for integer type,* `r` *for real type,* `z` *for complex type,* `l` *for logical type,* `c` *for character type and* `t` *for a derived data type appended with two underscore characters. The components shall consist of capitalized letters and numerals. The components shall be separated by a separator character.*

Example : ***l__PARAMETER_NAME***

Naming convention 18 `Local parameter:` *Shall have a one character prefix identifying the data type of the parameter,* `i` *for integer type,* `r` *for real type,* `z` *for complex type,* `l` *for logical type,* `c` *for character type and* `t` *for a derived data type. The components shall consist of capitalized letters and numerals. The components shall be separated by a separator character.*

Example : ***lPARAMETER_NAME***

Naming convention 19 `Global parameter:` *Shall have a prefix consisting of the module acronym appended with an underscore character, a character identifying the data type of the parameter,* `i` *for integer type,* `r` *for real type,* `z` *for complex type,* `l` *for logical type,* `c` *for character type and* `t` *for a derived data type, and the application acronym. The components shall consist of capitalized letters and numerals. The components shall be separated by a separator character.*

Example : ***MODEX_lfPARAMETER_NAME***

Variable names

Variables may be categorized according to their scope as follows:

- Local with respect to a procedure or local procedure. The scope of this variable is the procedure, but not a dummy argument. We make a further distinction between variables of this type.
    - A variable used for indexing purposes is referred to as an *index variable* or *local index variable* respectively. An index variable have no particular meaning, in relation to the algorithm, other than being an auxiliary enumeration variable. The type of an index variable is integer.
    - A variable used to express an entity in an algorithm are referred to as a *procedure local variable* or *local procedure local variable* respectively.
- Global with respect to a procedure. The scope of this variable is the procedure and the interface of the procedure. We will refer to this type of variable as a *dummy argument* or *local dummy argument* respectively.

- Local with respect to a module. The scope of this variable is the module. We will refer to this type of variable as a *local variable*.
- Global with respect to a module. The scope of this variable is potentially all modules. We refer to this type of variable as a *global variable*.

There is also another property of a variable that should be readily detectable, thus we introduce a categorization of variables independent of scope, as follows:

- Variable is a *scalar*.
- Variable is an *array*.

We now introduce a naming convention for each type of variable as defined above.

Naming convention 20 `Index variable:` *Shall have no prefix. The components shall have lower case letters and numerals. Separators shall be used in constructions.*

Example : ***index_name***

Naming convention 21 `Local index variable:` *Shall have no prefix. The components shall have lower case letters and numerals. Separators shall be used in constructions and it shall have a suffix consisting of two underscore characters.*

Example : ***index_name__***

Naming convention 22 `Procedure local variable:` *Shall have a one character prefix identifying the data type of the variable,* `i` *for integer type,* `r` *for real type,* `z` *for complex type,* `l` *for logical type,* `c` *for character type and* `t` *for a derived data type appended with the underscore character. The components shall have lower case letters and numerals. Separators shall be used in constructions.*

Example : ***i_variable_name***

Naming convention 23 `Local procedure local variable:` *Shall have a one character prefix identifying the data type of the variable,* `i` *for integer type,* `r` *for real type,* `z` *for complex type,* `l` *for logical type,* `c` *for character type and* `t` *for a derived data type appended with two underscore characters. The components shall have lower case letters and numerals. Separators shall be used in constructions and it shall have a suffix consisting of two underscore characters.*

Example : ***i__variable_name***

Naming convention 24 `Dummy argument:` *Shall have a one character prefix identifying the data type of the variable,* `i` *for integer type,* `r` *for real type,* `z` *for complex type,* `l` *for logical type,* `c` *for character type,* `t` *for a derived data type and* `p` *for procedure type appended with the underscore character. The components shall start with a capital letter and for the remaining have lower case letters and numerals and no separators.*

Example : ***i_VariableName***

Naming convention 25 *`Local dummy argument`: Shall have a one character prefix identifying the data type of the variable, `i` for integer type, `r` for real type, `z` for complex type, `l` for logical type, `c` for character type, `t` for a derived data type and `p` for procedure type appended with two underscore characters. The components shall start with a capital letter and for the remaining have lower case letters and numerals and no separators and it shall have a suffix consisting of two underscore characters.*

Example : ***i__VariableName***

Naming convention 26 *`Local variable`: Shall have a one character prefix identifying the data type of the variable, `i` for integer type, `r` for real type, `z` for complex type, `l` for logical type, `c` for character type and `t` for a derived data type. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : ***iVariableName***

We note that the local variable and the derived data type component are defined identically. Since the derived data type component is always associated with a derived data type no uniqueness problem is anticipated.

Naming convention 27 *`Global variable`: Shall have a prefix consisting of the module acronym appended with an underscore character and a character identifying the data type of the variable, `i` for integer type, `r` for real type, `z` for complex type , `c` for character type and `t` for a derived data type appended with the application acronym. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Example : ***MODEX_ifVariableName***

Naming convention 28 *`Scalar or array type variable and function (procedure)`: If the variable or function result is scalar no suffix shall be used (local index variables is the one exception). If the variable or function is an array a _ (underscore) suffix shall be used. This convention shall apply to all variables independent of scope.*

Example : ***i_scalar, i_array_, fr_scalar_valued_function, fr_array_valued_function_***

Fortran 90/95 coding style and syntax

In the IfeFEM 3.0 project we standardize to the Fortran 90/95 programming language in a strict sense. In Fortran 90/95 a number of new language features and alternative syntax have been added for a number of statements. To introduce a unified programming style in IfeFEM 3.0 we make restrictions with respect to alternative syntax. We denote these restrictions as *syntax conventions*. Furthermore we make *recommendations* concerning the use of new language features. There will always be conflicting concerns when making such decisions. To resolve such conflicts we have used the following priority list:

1. Modular code:

A modular code is characterized by a subdivision of a problem into well defined subtasks with well defined interfaces. A design based on this criterion may be realized by the use of modules, see section 3.5. The interface is defined through module procedures. New applications may rapidly be implemented by using modules. The risk of making errors is greatly reduced and the productivity will increase.

2. Scalable code:

A software code is referred to as *scalable* when it can be applied on any sized problem. This property simplifies the use of a procedure library like IfeFEM 3.0. It also eliminates the source of numerous errors.

3. Legible code:

Write the code in such a way that it may be understood by others. Use indentation, see section 9.1, comments and a naming convention, see section 2.

4. Efficient code:

Reduce the computation times. Use the best algorithms.

5. Compact code:

Reduce the number of source lines to edit. The use of array syntax, see section 3.4.2, will result in compact code. Multiple statements on one source line is advised against.

However, in the first section we introduce the style conventions that define the layout of the Fortran 90/95 source code. In the next section we make an attempt to reduce the Fortran 90/95 syntax redundancy by introducing syntax conventions. Then we have devoted a section to general recommendations concerning the new features in Fortran 90/95. The single most important new feature in Fortran 90/95 for the IfeFEM 3.0 project is the module concept. A section is used for recommendations for coding a module according to the principles of data hiding and rudimentary object orientation. In the last section we present the features that have been banned in Fortran 90/95.

Style conventions

To produce legible source code we introduce conventions regarding the layout of the code. We will refer to these conventions as *style conventions*.

1.1.1 Source form

With Fortran 90/95 we have a choice between two source forms - fixed form (Fortran 77) and free form (Fortran 90/95). The free form format offers a number of improvements. For details see [1]. To make use of this we adopt the convention:

Style convention 1 *We shall use the free form source format. The maximum source line length permitted is 132 characters.*

Fortran 90/95 allows a line length of up to 132 characters, however this could cause problems when viewed on older terminals, or if print outs have to be obtained on A4 paper. In the free form source format multiple Fortran 90/95 statements per source line is allowed. The Fortran 90/95 statements must be separated by `;` - semicolon. This leads to compressed code, but reduces legibility. Since legibility has a higher priority we adopt the convention:

Style convention 2 *There shall be only one Fortran 90/95 statement per source line. Thus, in any Fortran 90/95 project the use of* `;` *- semicolon is advised against.*

1.1.2 Fortran 90/95 keywords and intrinsic procedures

To ease the reading of programming code it is a good practice to emphasize the language keywords. This is particularly important with Fortran 90/95 since the keywords here are not protected. We also take a stand with respect to obsolecent and redundant features. To accommodate this in the IfeFEM 3.0 project we adopt the following convention:

Style convention 3 *Fortran keywords and intrinsic procedures shall be written in capital letters. We avoid the use of obsolescent and redundant features.*

1.1.3 Entity names

A source code is easier to read if the Fortran 90/95 entities such as variable names, procedure names, type names and module names have meaningful names. Thus we introduce the convention:

Style convention 4 *Named Fortran 90/95 entities shall have meaningful names in English. Recognized abbreviations are acceptable as a means of preventing entity names getting too long.*

To simplify and unify the named entities in Fortran 90/95 a strict naming convention has been introduced, see section 2. To enforce this we introduce the convention:

Style convention 5 *Names for Fortran 90/95 entities shall be given according to the naming convention defined in section 2.*

1.1.4 Precompiler

A precompiler or preprocessor is a tool that simplifies source code maintenance. The precompiler is capable of macro substitution, conditional compilation and inclusion of named files. Precompiler directives to accomplish these tasks must be included in the source file. To produce a valid Fortran 90/95 source file we must filter it through the precompiler. A number of precompilers may be used. The standard C preprocessor would be a candidate. However, the C processor is not completely standardized across the different platforms. **f90ppr** is a public domain Fortran 90/95 precompiler, see section 9.1. With the source code available we can compile a version of **f90ppr** for the platforms of interest. In this way we ensure that the precompiler is completely standardized. A further advantage with **f90ppr** is that it may also be used as a source code formatter. Based on these observation we introduce the convention:

Style convention 6 *We shall use **f90ppr** as our precompiler. We shall use the precompiler directives defined in the **f90ppr** documentation. Source code files containing precompiler directives shall have the extension `.fpp`.*

1.1.5 Indentation

To improve the legibility of the source code we use indentation. The Fortran 90/95 precompiler **f90ppr** , see section 9.1, is also a pretty printer. The precompiler **f90ppr** produce a well balanced indentation. Thus, to enforce identical indentation for all source files we introduce the convention:

Style convention 7 *The source code shall be run through the precompiler **f90ppr** to produce a standardized indentation in the source code.*

1.1.6 Tab characters

The use of tab characters may easily ruin the source code when ported to another platform. The editor on the new platform may interpret the tab characters differently. To avoid this potential problem we introduce the convention:

Style convention 8 *We shall not use tab characters in any source file.*

1.1.7 Source code administration

As a general rule we adopt the convention:

Style convention 9 *Each program unit shall be stored in a separate file.*

However, source files may become impractically large to work with in a text editor. In practice this is most likely to happen with modules containing a large number of module procedures. To avoid this problem we adopt the convention:

Style convention 10 *Module procedures with more than 100 lines of source may be stored in a separate file. We use the Fortran 90/95 `INCLUDE` statement to include the module procedure in the module source file.*

Where files shall be located in the IfeFEM 3.0 directory hierarchy is explained in section 8.

Syntax conventions

Fortran 90/95 introduces an alternative syntax for a number of features in the language. Where the new syntax is considered better (by us) we restrict ourselves to this syntax. Many new features have been added to the language. For a complete list of the new features see [1]. We encourage the use of the new features where these improve either legibility, efficiency , modularity , data hiding or object oriented design possibilities.

1.1.8 Declaration of named entities

In Fortran 90/95 variables and dummy arguments may have attributes. To reflect this new and powerful feature we have adopted the following convention:

Syntax convention 1 *In the declaration of named entities of type `INTEGER`, `REAL`, `COMPLEX`, `CHARACTER`, `LOGICAL` and `TYPE(type-name)` we shall use the syntax:*

```
    type, attribute :: entity-list
```

For detailed definitions of `type`, `attribute` and `entity-list` see [1]. With the exception of `PUBLIC` and `PRIVATE`, we shall not use the statement form of type attributes. The `PUBLIC` and `PRIVATE` statements will be used to define the accessibility of module procedures and operators.

Accepting syntax convection 1 we have two alternatives when declaring arrays. We may do it with the attribute `DIMENSION` or directly by appending the rank to the variable name. The latter choice will result in more compact code with no reduction in legibility. Thus we prefer this and introduce the convention:

Syntax convention 2 *Declaration of arrays shall be done by appending the shape/rank definition to the variable name, and not by the use of the `DIMENSION` attribute*

1.1.9 Complete function definition

The definition of a `FUNCTION` in Fortran 90/95 may have different forms. A new syntax is required in the definition of a recursive `FUNCTION`. Due to this we will only accept the new syntax in the definition of a `FUNCTION`. Thus we adopt the following convention:

Syntax convention 3 *We shall use the following syntax in the definition of a `FUNCTION`.*

```
[prefix] FUNCTION function-name( [dummy-argument-list] )
RESULT( result-name )
```

The only `prefix` allowed is `RECURSIVE`. The type of the `FUNCTION` is determined by the type definition of `result-name`. We define the type of `result-name` first and then the `dummy-argument-list`.

1.1.10 Complete form of end statement

The `END` statement in the definition of a program , module , procedure and derived data type has alternative forms. We only allow the complete form and thus introduce the convention:

Syntax convention 4 *We shall use the complete form of the `END` statement. The complete form of the `END` has the following syntax:*

```
END entity entity-name
```

`entity` is one of `PROGRAM`, `MODULE`, `SUBROUTINE`, `FUNCTION` and `TYPE`. The name of the entity is referenced by `entity-name`. In the Fortran 90/95 standard no `entity-name` is allowed for `INTERFACE`. In the Fortran 95 standard this restriction is removed.

1.1.11 Relational operators

An alternative set of relational operators have been introduced in Fortran 90/95. Since this set increases the readability we adopt the convention:

Syntax convention 5 *We shall use the relational operators <, <=, ==, /=, > and >= instead of `.LT.`, `.LE.`, `.EQ.`, `.NE.`, `.GT.`, `.GE.`.*

1.1.12 Control statements

The `SELECT CASE` construct is new to the language. This construct improves legibility and is more efficient than an alternative `IF THEN ELSE` construct would be. The `SELECT CASE` construct is restricted to `INTEGER` and `CHARACTER` options. For these situations we adopt the convention:

Syntax convention 6 *We shall use the `SELECT CASE` construct to express a multiple choice situation involving an `INTEGER` or `CHARACTER` option.*

A general while loop has been introduced in Fortran 90/95. The loop control statements `CYCLE` and `EXIT` may be used with this loop construct. These new statements has made the `DO WHILE` statement redundant. We therefore introduce the convention:

Syntax convention 7 *We shall not use the `DO WHILE` loop. We use the more general while loop instead:*

```
DO
  IF ( logical_expression ) EXIT
  .
  .
END DO
```

In Fortran 90/95 it is possible to assign a name to a `DO` loop. This is a feature that may be used to enhance the legibility. We therefore introduce the conventions:

Syntax convention 8 *We shall assign a name to any large and complicated loops to clarify which statements are associated with the loop.*

Syntax convention 9 *Assign names to all nested loops so that they will be easier to understand and debug*

Syntax convention 10 *We shall use loop names with `CYCLE` or `EXIT` statements to make sure that the statement affect the proper loop.*

1.1.13 Character strings

A new delimiter, `"`, for character strings has been introduced in Fortran 90/95. Since this is more visible, we prefer the use of it.

Syntax convention 11 *We shall use the token `"` to delimit character strings.*

Banned features

We ban Fortran 90/95 features based on the following criteria:

1. Features advised against in the Fortran 90/95 standard.
2. Features made redundant by the Fortran 90/95 standard.
3. Features whose use is deemed to be bad programming practice as they can degrade the maintainability of code.

Based on these criteria we introduce the following convention:

Syntax convention 12 *Explicitly banned features are:*

1. *`COMMON` blocks - use `MODULE`'s instead.*
2. *`EQUIVALENCE` - use `POINTERS` or derived data types instead.*
3. *Assigned and computed `GO TO`'s - use the `CASE` construct instead.*
4. *`DO WHILE` loops - use the general while loop `DO` instead.*
5. *Arithmetic `IF` statements - use the block `IF` construct instead.*
6. *Labels (only one allowed use).*
   - *Labelled `DO` constructs - use End `DO` instead.*
   - *I/O routine's `END =` and `ERR =` use `IOSTAT` instead.*

- o *`FORMAT` statements: use Character parameters or explicit format specifiers inside the Read or Write statement instead.*
- o *`GO TO`*
- o *The only recognized use of `GO TO`, indeed of labels, is to jump to the error handling section at the end of a routine on detection of an error. The jump must be to a `CONTINUE` statement and the label used must be 9999. Evens so, it is recommended that this practice be avoided.*
- o *Any other use of `GO TO` can probably be avoided by making use of `IF`, `CASE`, `DO`, `EXIT` or `CYCLE` statements. If a `GO TO` really has to be used, then clearly comment it to explain what is going on and terminate the jump on a similarly commented `CONTINUE` statement.*

7. *`PAUSE`*
8. *`ENTRY` statements: - a subprogram may only have one entry point.*
9. *Functions with side effects i.e. functions that alter variables in their argument list or in modules used by the function; or one that performs I/O operations. This is very common in C programming, but can be confusing. Also, efficiencies can be made if the compiler knows that functions have no side effects. High Performance Fortran, a variant of Fortran 90/95 designed for massively parallel computers, will allow such instructions.*
10. *Implicitly changing the shape of an array when passing it into a subroutine. Although actually forbidden in the standard it was very common practice in Fortran 77 to pass 'n' dimensional arrays into a subroutine where they would, say, be treated as a 1 dimensional array. This practice, though banned in Fortran 90/95, is still possible with external routines for which no `INTERFACE` block has been supplied. This only works because of assumptions made about how the data is stored: it is therefore unlikely to work on a massively parallel computer. Hence the practice is banned.*

General recommendations

So far we have limited experience with the new features of Fortran 90/95. However, we have enough to make some general recommendations on their use.

1.1.14 Pointers

Pointers are a new and powerful feature of Fortran 90/95. However, there is one subtlety associated with pointers in Fortran 90/95. Their initial (after declaration) state is undefined. Undefined pointers are not allowed in the `ASSOCIATED` intrinsic function. Thus, to always be able to inquire the state of a pointer makes sure to initialize (`NULLIFY`) it. Based on this we make the following recommendation:

Recommendation 1 *Pointers should always be initialized with the intrinsic functions `NULLIFY` or `NULL`.*

1.1.15 Array syntax

Array syntax is one of the most powerful new features of Fortran 90/95. Using array syntax in the implementation of algorithms makes the code compact and more legible. We therefore introduce the convention:

Recommendation 2 *We use array syntax whenever possible.*

1.1.16 Intrinsic procedures

A number of new intrinsic procedures have been specified and added to Fortran 90/95, for a complete list see [1]. Many of them are useful in numerical modelling work, such as the vector and matrix multiplication functions and the numeric inquiry functions. There also exists commercially available libraries with optimized and parallel versions of some of them (`DOT_PRODUCT` and `MATMUL`). A code using intrinsic procedures will be regarded as standard conforming. To encourage the use of the intrinsic procedures we introduce the convention:

Recommendation 3 *We use intrinsic procedures whenever possible*

1.1.17 Parameterization of intrinsic types

Parameterized intrinsic types is a new feature in Fortran 90/95 that permits processors to support short integers, very large character sets, more than two precisions for real and complex and packed logicals. This is a very attractive feature since we no longer need to maintain a `REAL` and `DOUBLE PRECISION` version of a program separately. Taking full use of this feature the conditional compilation overhead is reduced. Thus we introduce the convention:

Recommendation 4 *We make use of the possibility to parameterize the intrinsic types.*

1.1.18 Derived data types

To express algorithms naturally we often need more advanced data types than the intrinsic types. With the introduction of derived data types in Fortran 90/95 we can ourselves construct the data types which we find suitable. The use of derived data types enhances the modularity and simplifies the maintenance of a program. We therefore adopt the convention:

Recommendation 5 *We use user derived data types whenever they serve a purpose.*

User derived types with dynamic size components can only be implemented with the POINTER attribute in Fortran 90/95. This may have a serious performance penalty. This restriction is removed in FORTRAN 2003.

1.1.19 Internal and recursive procedures

Internal procedures are a generalization of the statement functions in Fortran 77. Since their use produce more readable code we encourage their use instead of statement functions. Recursive procedures are a new feature in Fortran 90/95. We use this feature to implement algorithms including either direct or indirect recursion.

Recommendation 6 *We use internal procedures instead of statement functions, and we use recursive procedures in the implementation of recursive algorithms.*

1.1.20 Procedures whit optional dummy arguments

Sometimes dummy argument lists become large and contain arguments that are not required. In this situation the use of `OPTIONAL` arguments may be useful. We will not encourage extensive use of this feature. Rather, we encourage the use of derived types to shorten the argument lists. It is therefore natural to introduce the convention:

Recommendation 7 *We use optional arguments where this is natural.*

1.1.21 Dynamic memory

Dynamic storage is a new feature in Fortran 90/95. It makes the task of conserving memory much simpler. In the potentially memory hungry applications we are aiming for this is particularly good news. One of the most serious problems with previous versions of IfeFEM has been the lack of scalability. With dynamic memory scalability (at least up to hardware memory limits) can be achieved. This calls for the following recommendation:

Recommendation 8 *We make use of dynamic memory to enforce scalability of any Fortran 90/95 application.*

Care must be taken, however, as there is potential for inefficient memory usage, particularly in parallelized code. For example heap fragmentation can occur if space is allocated by a lower level routine and then not freed before control is passed back up the calling tree. There are three ways of obtaining dynamic memory in Fortran 90/95:

Automatic arrays:

> These are arrays initially declared within a subprogram whose extents depend upon variables known at runtime e.g. variables passed into the subprogram via its argument list.

Pointer arrays:

> Array variables declared with the POINTER attribute may be allocated space at run time by using the ALLOCATE command.

Allocatable arrays:

> Array variables declared with the ALLOCATABLE attribute may be allocated space at run time by using the ALLOCATE command. However, unlike pointers, allocatables are not allowed inside derived data types.

We introduce the following recommendations for the safe use of dynamic memory:

Recommendation 9

- *Use automatic arrays in preference to the other forms of dynamic memory allocation when arrays are of smaller size (1-100 array elements).*
- *Space allocated by pointers and allocatable arrays must be explicitly freed using the DEALLOCATE statement.*
- *In a given program unit do not repeatedly ALLOCATE space, DEALLOCATE it and then ALLOCATE a larger block of space. This will almost certainly generate large amounts of unusable memory.*
- *Always test the allocation state of a varaiable before allocating space with ALLOCATE. Use ALLOCATED (inquiry intrinsic function) for allocatable arrays and ASSOCIATED (inquiry intrinsic function) for pointer arrays.*
- *Always test the success of a dynamic memory allocation and deallocation. The ALLOCATE and DEALLOCATE statements have an optional argument to let you do this.*

Module recommendations

The module is a new feature in Fortran 90/95. Due to its potential usefulness in the IfeFEM 3.0 project we treat it separately. The module is a new program unit which generalizes and outdates the use of `COMMON` . It is the Fortran 90/95 parallel to the `C++` class concept. Within the scope of the module the user may define global and local data and procedures processing this data. The intended use of modules was for the construction of procedure libraries. Thus the module is precisely the building block we need in the IfeFEM 3.0 project. Since IfeFEM 3.0 is meant to be a procedure library we make the recommendation:

Recommendation 10 *We use the module as the basic building block in our Fortran 90/95 applications.*

In the module head we specify the data needed by the module. The default access to this data is PUBLIC. This means that it may be accessed by procedures in the module, other modules, external procedures and the main program by so called *use association*. This default access may have some undesirable consequences.

- Unnecessary reduction of the name space.
- The access to module data outside the module is a potential problem. The maintenance is no longer restricted to the module itself. The state of the module may be changed from the outside. Thus, the protection of the module data is lost.
- Indirect use association.

To reduce the maintenance costs of the module it should be self-contained and the exchange of module data should be done through module procedures with a well defined interface. Thus, to be in accordance with the modern principle of *data hiding* or *data encapsulation* we make the following recommendation:

Recommendation 11 *The default accessibility of module variables, data type definitions, procedures and operators should be declared explicitly. The default access of any module should be private.*

In the *implementation part* of the module (`CONTAINS` part) we define the module procedures. Module procedures have access to all the data defined in the module head. The access to the module procedures may be limited in the same way as the data in the module head. With default access private the module procedures needed outside the module must be explicitly declared public. This calls for the recommendation:

Recommendation 12 *Entities in any module with a global scope should be declared public explicitly.*

Public entities from other modules can be made accessible in a current module by `USE` association. It is good practice to restrict the accessibility to only those entities needed by the current module. This may be accomplished by using the `ONLY` keyword with the `USE` statement. Based on this comment we introduce the recommendation:

Recommendation 13 *We should use the `USE,ONLY` statement to specify which variables, data type definitions, procedures and operators defined in the module are to be accessible in the `USE'`ing module.*

To allow module data to be shared between program units by use association they must have the `SAVE` attribute. Thus we introduce the recommendation:

Recommendation 14 *Shared module data must be declared with the `SAVE` attribute.*

To enforce strong typing of module data and in all module procedures we introduce the recommendation:

Recommendation 15 *The head of any module should contain the `IMPLICIT NONE` statement.*

If the intent of dummy arguments in module procedures are defined, the compiler will provide better optimization. We therefore recommend:

Recommendation 16 *The INTENT of dummy arguments in any module procedures should be defined if possible. In the case of POINTER arguments the intent may not be defined.*

The module is also the natural place for definitions of extended assignment operators, overloaded operators, general operators and generic procedures. Some general comments may be appropriate when the module contains derived type definitions.

To provide proper memory allocation and initialization of the derived type with basename `Type` we should supply a *constructor*. The purpose of the constructor is to allocate memory, initialize derived type components etc. We introduce the recommendation:

Recommendation 17 *A constructor for a derived type should be provided. The constructor shall be implemented as a subroutine. The first dummy argument of the subroutine shall be a pointer to the derived type `Type`. The constructor shall have the basename `New`. If more derived types are defined, the basename of the constructors should be `New` appended with the basename of the derived type. To be in accordance with a single constructor per module we provide a generic constructor with the basename `New`. If the derived type `Type` is public the constructor, `New`, must be public.*

To provide proper memory deallocation of the derived type `Type` we should provide a *destructor* . The purpose of the destructor is to deallocate memory etc. We introduce the recommendation:

Recommendation 18 *A destructor for a derived type should be provided. The destructor shall be implemented as a subroutine. The first dummy argument of the subroutine shall be a pointer to the derived type `Type`. The destructor shall have the basename `Delete`. If more derived types are defined, the basename of the destructors should be `Delete` appended with the basename of the derived type. To be in accordance with a single destructor per module we provide a generic destructor with the basename `Delete`. If the derived type `Type` is public the destructor, `Delete`, must be public.*

Fortran 90/95 doesn't support automatic printing of derived data types. This calls for the inclusion of a print function for the derived type. Such a function is very handy in a debug situation. Based on this comment we make the recommendation:

Recommendation 19 *A print subroutine for a derived type should be provided. The first dummy argument of the print subroutine shall be a pointer to the derived type `Type`. The print subroutine shall have the basename `Print`. If more derived types are defined, the basename of the print subroutine shall be `Print` appended with the basename of the derived type. To be in accordance with a single print function per module we provide a generic print function with the basename `Print`. If the derived type `Type` is public the print function, `Print`, must be public.*

When a derived type, with basename `Type`, has pointer components the generic assignment operator may not have the desired effect. The generic assignment operator for derived data types implies pointer assignment for pointer components. If you actually need the assignment operator to produce a copy, a new assignment operator must be supplied for the derived type. We introduce the recommendation:

Recommendation 20 *An assignment operator for a derived type should be provided if the derived type `Type` contains pointer components and the generic assignment operator is not satisfactory. The assignment operator subroutines shall have the name `aType`. If the derived type `Type` is public the assignment operator, =, must be public.*

In practical work with IfeFEM 3.0, one of the most beneficial uses of modules is the automatic type checking of module procedures. In section 4 we give you an example of a module implemented after the principles above.

Templates

To accommodate the implementation of the IfeFEM 3.0 syntax conventions, recommendations and naming conventions introduced above we have made templates for the basic building blocks of Fortran 90/95. We have templates for program units and operators. We hope that the templates will be a useful starting point for the implementation of new modules in IfeFEM 3.0, and make it easier to take advantage of the more advanced new features of Fortran 90/95. The use of templates, we believe, will also encourage a unified programming style.

We now present the templates and add comments to statements of particular interest. Text inside <> brackets must be replaced with appropriate text by the user. The comments are included in the source code in the form of ordinary Fortran 90/95 comments preceded with a single `!` sign. For the time being ignore comments starting with `!!`. Their meaning will be explained in section 6.

The templates are meant to be a starting point for the development of new Fortran 90/95 program units and operators. Just copy the relevant template file from the template directory in the IfeFEM 3.0 directory structure, see section 8, and you are ready to go. Then edit the text inside <> brackets and add the necessary code. The relevant comments ending with `:` may be left in the code as general comments, see section 5.2. Remember to delete irrelevant comments.

In the first section we present templates for program units. Then a section is devoted to templates for operators.

Program units

1.1.22 Program

Below we find the template for the program unit program:

```
!+ <A one line description of this program>
!----------------------------------------------------------------------
---------
!! @description
!!  <Say what this program does>
!!
!! @method
!!  <Say how it does it: include references to external documentation>
!!  <If this routine is divided into sections, be brief here,
!!   and put method comments at the start of each section>
!!
!! @input_files
!!  <Describe these, and say in which routine they are read>
!!
!! @utput_files
!!  <Describe these, and say in which routine they are written>
!!
!! @owner <Name of person responsible for this code>
!!
!! @history
!!  <version>  <date>  <responsible>  <comment>
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------

PROGRAM <NameOfProgram>

! Declarations:

! Modules used:

!! <A one line description of module and the purpose of association>

USE <ModuleName>, ONLY : &
! Imported Parameters:

! Imported Type Definitions:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):

! Imported Array Variables with intent (out):

! Imported Procedures:

! Repeat from Use for each module...

IMPLICIT NONE

! Include statements:

! Declarations must be of the form
! <type>, attributes ::  <VariableName> !! Description/ purpose of
variable

! Local parameters:

! Local scalars:

! Local arrays:

! Define code body of the program.

CONTAINS

! Define internal procedures contained in this program.

END PROGRAM <NameOfProgram>
```

1.1.23 Module

Below we find the template for the program unit module:

```
!+ <A one line description of this module>
!----------------------------------------------------------------------
---------
!! @description
!!  <Say what this module is for>
!!
!! @owner <Name of person responsible for this code>
!!
!! @procedure_list
!!
!! @history
!!  <version>  <date>  <responsible>  <comment>
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------

MODULE <ModuleName>

! Modules used:

!! <A one line description of module and the purpose of association>

USE <ModuleName>, ONLY : &
! Imported Parameters:

! Imported Type Definitions:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):

! Imported Array Variables with intent (out):

! Imported Procedures:

! Repeat from USE for each module...

! Impose strong typing of module data and all module procedures:

IMPLICIT NONE

! Make the default access of the module private:

PRIVATE

! Declarations must be of the form
! <type>, attributes ::  <VariableName> !! Description/ purpose of
variable
! PUBLIC :: <ModuleProcedureNameList>

! Global (i.e. public) Declarations:
! Global Parameters:

! Global Type Definitions:
```

```
! Global Scalars:

! Global Arrays:

! Global Operators:

! Global Module Subroutine:

! Global Module Functions:

! Local (i.e. private) Declarations:
! Local Parameters:

! Local Type Definitions:

! Local Scalars:

! Local Arrays:

! Operator definitions:
!   Define new operators or overload existing ones.

! Generic functions definitions:
!   Define the generic functions.

CONTAINS

! Module operator procedures:
! Define operator procedures contained in this module.

! Module procedures:
! Define procedures contained in this module.

! Comment: If a module procedure contains more than 100
!          lines of source code store it in a sparate file.
!          Include it in the module with the Fortran90
!          INCLUDE statement.

END MODULE <ModuleName>
```

1.1.24 Subroutine

Below we find the template for the program unit subroutine:

```
!+ <A one line description of this subroutine>
!-----------------------------------------------------------------------
---------
!! @description
!!  <Say what this routine does>
!!
!! @method
!!  <Say how it does it: include references to external documentation>
!!  <If this routine is divided into sections, be brief here,
!!   and put method comments at the start of each section>
!!
!! @owner <Name of person responsible for this code>
!!
!! @history
!!  <version>  <date>  <responsible>  <comment>
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------

SUBROUTINE <SubroutineName> &
           (<InputArguments, InOutArguments, OutputArguments>)

! Declarations:

! Modules used:

!! <A one line description of module and the purpose of association>

USE <ModuleName>, ONLY : &
! Imported Parameters:

! Imported Type Definitions:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):

! Imported Array Variables with intent (out):

! Imported Procedures:

! Repeat from USE for each module...

IMPLICIT NONE
! This statement is not necessary if the subroutine is a module
procedure.

! Include statements:

! Declarations must be of the form
! <type>, attributes ::  <VariableName> !! Description/ purpose of
variable

! Subroutine arguments:
! Scalar arguments with intent(in):

! Array  arguments with intent(in):

! Scalar arguments with intent(inout):

! Array  arguments with intent(inout):

! Scalar arguments with intent(out):

! Array  arguments with intent(out):

! Local parameters:

! Local scalars:
```

```
! Local arrays:

! Define code body of the subroutine.

CONTAINS

! Define internal procedures contained in this subroutine.

END SUBROUTINE <SubroutineName>
```

1.1.25 Function

Below we find the template for the program unit function:

```
!+ <A one line description of this function>
!----------------------------------------------------------------------
---------
!! @description
!!  <Say what this function does>
!!
!! @method
!!  <Say how it does it: include references to external documentation>
!!  <If this routine is divided into sections, be brief here,
!!   and put method comments at the start of each section>
!!
!! @owner <Name of person responsible for this code>
!!
!! @history
!!  <version>  <date>   <responsible> <comment>
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------

FUNCTION <FunctionName> (<InputArguments>) &
         RESULT (<ResultName>)

! Declarations:

! Modules used:

!! <A one line description of module and the purpose of association>

USE <ModuleName>, ONLY : &
! Imported Parameters:

! Imported Type Definitions:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):

! Imported Array Variables with intent (out):
```

```
! Imported Procedures:

! <Repeat from Use for each module...>

IMPLICIT NONE
! This statement is not necessary if the function is a module
procedure

! Include statements:

! Declarations must be of the form
! <type>, attributes ::  <VariableName> !! Description/ purpose of
variable

! Function arguments:
! Scalar arguments with intent(in):

! Array  arguments with intent(in):

! Result argument:

! Local parameters:

! Local scalars:

! Local arrays:

! Define code body of the function.

CONTAINS

! Define internal procedures contained in this function.

END FUNCTION <FunctionName>
```

Operators

1.1.26 Assignment

Below we find the template for the assignment operator subroutine:

```
!+ <A one line description of this assignment subroutine>
!-------------------------------------------------------------------
---------
!! @description
!!  <Say what this assignment subroutine does>
!!
!! @method
!!  <Say how it does it: include references to external documentation>
!!  <If this routine is divided into sections, be brief here,
!!   and put method comments at the start of each section>
!!
!! @owner <Name of person responsible for this code>
!!
!! @history
!!  <version>  <date>  <responsible>  <comment>
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------

SUBROUTINE <AssignmentSubroutineName> &
           (<LhsInputArgument, RhsOutputArgument>)

! Declarations:

! Modules used:

!! <A one line description of module and the purpose of association>

USE <ModuleName>, ONLY : &
! Imported Parameters:

! Imported Type Definitions:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):

! Imported Array Variables with intent (out):

! Imported Procedures:

! Repeat from USE for each module...

IMPLICIT NONE
! This statement is not necessary if the subroutine is a module
procedure.

! Include statements:

! Declarations must be of the form
! <type>, attributes ::  <VariableName> !! Description/ purpose of
variable

! Subroutine arguments:
! May not contain the OPTIONAL attribute. The intent of the Lhs
argument
! must be OUT and for the Rhs argument it must be IN.

! Scalar arguments with intent(in):

! Array  arguments with intent(in):

! Scalar arguments with intent(out):

! Array  arguments with intent(out):

! Local parameters:

! Local scalars:

! Local arrays:
```

```
! Define code body of the subroutine.

CONTAINS

! Define internal procedures contained in this subroutine.

END SUBROUTINE <AssignmentSubroutineName>
```

1.1.27 General

There are two kinds of operators, *overloaded* and *user defined* . An overloaded operator is an extension of one of the Fortran 90/95 intrinsic operators. The operator symbol in this case is the intrinsic operator symbol. The user defined operator is defined freely by the user. The operator symbol in this case will be `.OperatorName.`, where `OperatorName` is replaced by the users wish. Below we find the template for the operator function:

```
!+ <A one line description of this operator>
!-----------------------------------------------------------------------
---------
!! @description
!!  <Say what this operator does>
!!
!! @method
!!  <Say how it does it: include references to external documentation>
!!  <If this routine is divided into sections, be brief here,
!!   and put method comments at the start of each section>
!!
!! @owner <Name of person responsible for this code>
!!
!! @history
!!  <version>  <date>  <responsible>  <comment>
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!-----------------------------------------------------------------------
---------

FUNCTION <OperatorFunctionName> (<InputArgument1>,<InputArgument2>) &
         RESULT (<OperatorResultName>)

! Declarations:

! Modules used:

!! <A one line description of module and the purpose of association>

USE <ModuleName>, ONLY : &
! Imported Parameters:

! Imported Type Definitions:

! Imported Scalar Variables with intent (in):

! Imported Scalar Variables with intent (out):

! Imported Array Variables with intent (in):
```

```
! Imported Array Variables with intent (out):

! Imported Procedures:

! <Repeat from Use for each module...>

IMPLICIT NONE

! Include statements:

! Declarations must be of the form
! <type>, attributes ::  <VariableName> !! Description/ purpose of
variable

! Function arguments:
! May not contain the OPTIONAL attribute. If the function defines a
! unary operator only one input argument is needed. A binary operator
! will need two input arguments. The intent of the arguments must be
IN.

! Scalar arguments with intent(in):

! Array  arguments with intent(in):

! Result argument:

! Local parameters:

! Local scalars:

! Local arrays:

! Define code body of the operator function.

CONTAINS

! Define internal procedures contained in this operator function.

END FUNCTION <OperatorFunctionName>
```

Examples

In the following sections we will find an example implementations of the program units and operators. These examples are base on the use of templates and the conventions defined in this standard.

1.1.28 Program

Below we find an example implementation of a program:

```
!+ This is a simple example program to illustrate use of a linked list
module.
!-----------------------------------------------------------------------
---------
!! @description
!!  This is a simple driver program to test the linked list extension
of the
!!  base module MODULE_NAME?. We test all the operations defined for
linked
!!  lists.
!!
!! @owner Magne Rudshaug
!!
!! @history
!!  0.10  07.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!-----------------------------------------------------------------------
---------
!
PROGRAM pTestLinkedList
!
! Declarations:
!
! Modules used:
!
!! A container module for the base type GRID_dfGrid.
      USE mfGrid
!
      IMPLICIT NONE
!
! Local parameters:
!
! Local scalars:
      INTEGER :: i_err !! Error status identifier.
      INTEGER :: i_number_of_members !! Number of members in grid
linked list.
      INTEGER :: i_size_of_grid !! The size of the grid.
      TYPE (GRID_dfGridLl), POINTER :: t_gridll !! The linked list of
grids.
      TYPE (GRID_dfGridLlM), POINTER :: t_gridllm !! A pointer to a
linked list
                                                   !! member.
      TYPE (GRID_dfGrid), POINTER :: t_grid !! The base type of the
linked list
                                             !! member.
      CHARACTER (LEN=GRID_ifNAME_LENGTH) :: c_name_of_member !! Name
of linked
                                                              !! list
member.
!
! Local arrays:
!
! Initaializes the pointer to the grid linked list.
      NULLIFY (t_gridll)
!
! Initializes the grid linked list.
      CALL GRID_fNew (t_gridll)
!
! Append some grids to the linked list.
      CALL GRID_fAppend (t_gridll, "ALSPEN grid", i_err)
      CALL GRID_fAppend (t_gridll, "ALSIM grid", i_err)
      CALL GRID_fAppend (t_gridll, "STEELTEMP grid", i_err)
      CALL GRID_fAppend (t_gridll, "STABILITY grid", i_err)
      CALL GRID_fAppend (t_gridll, "WELDSIM grid", i_err)
!
! Print the grids in the linked list.
      i_err = GRID_fifPrint (t_gridll)
!
! Count the number of members in the list.
      i_number_of_members = GRID_fifCount (t_gridll)
      WRITE (*,*) "Number of members in grid linked list:",
i_number_of_members
!
! Remove a grid from the linked list.
      CALL GRID_fRemove (t_gridll, "STEELTEMP grid", i_err)
!
! Print the grids in the linked list.
      i_err = GRID_fifPrint (t_gridll)
!
! Count the number of members in the list.
      i_number_of_members = GRID_fifCount (t_gridll)
      WRITE (*,*) "Number of members in grid linked list:",
i_number_of_members
!
! Append the STEELTEMP grid to the linked list again!
      CALL GRID_fAppend (t_gridll, "STEELTEMP grid", i_err)
!
! Print the grids in the linked list.
      i_err = GRID_fifPrint (t_gridll)
!
! Count the number of members in the list.
      i_number_of_members = GRID_fifCount (t_gridll)
      WRITE (*,*) "Number of members in grid linked list:",
i_number_of_members
!
! Getting a pointer to a linked list member.
      t_gridllm => GRID_ftfPointerMember (t_gridll, "STABILITY grid")
      CALL GRID_fName (t_gridllm, c_name_of_member, i_err)
      WRITE (*,*) "Name of linked list member : ", c_name_of_member
!
! Getting a pointer to a linked list member base type.
      t_grid => GRID_ftfPointerBase (t_gridll, "ALSPEN grid")
      i_size_of_grid = 3
      CALL GRID_fNew (t_grid, i_size_of_grid)
      CALL GRID_fPrint (t_grid)
      NULLIFY (t_grid)
!
! Copying the ALSPEN grid twice.
      CALL GRID_fCopy (t_gridll, "ALSPEN grid", "ALSIM grid", i_err)
      t_gridllm => GRID_ftfPointerMember (t_gridll, "ALSIM grid")
      CALL GRID_fPrint (t_gridllm)
      CALL GRID_fCopy (t_gridll, "ALSPEN grid", "PREAL grid", i_err)
      t_gridllm => GRID_ftfPointerMember (t_gridll, "PREAL grid")
      CALL GRID_fPrint (t_gridllm)
      NULLIFY (t_grid)
!
! Printing the current linked list.
      WRITE (*,*) "Printing the contents of the linked list!"
      WRITE (*,*)
      CALL GRID_fPrint (t_gridll)
!
```

```
! Getting a pointer to a linked list member base type.
      t_grid => GRID_ftfPointerBase (t_gridll, "STEELTEMP grid")
      i_size_of_grid = 4
      CALL GRID_fNew (t_grid, i_size_of_grid)
      CALL GRID_fPrint (t_grid)
      NULLIFY (t_grid)
      CALL GRID_fCopy (t_gridll, "STEELTEMP grid", "ALSPEN grid",
i_err)
      WRITE (*,*) "Printing the contents of the linked list!"
      WRITE (*,*)
      CALL GRID_fPrint (t_gridll)
!
! Remove two grids.
      WRITE (*,*) "Removes: STEELTEMP and STABILITY"
      CALL GRID_fRemove (t_gridll, "STEELTEMP grid", i_err)
      CALL GRID_fRemove (t_gridll, "STABILITY grid", i_err)
      CALL GRID_fPrint (t_gridll)
      i_number_of_members = GRID_fifCount (t_gridll)
      WRITE (*,*) "Number of members in grid linked list:",
i_number_of_members
!
! Delete all grids.
      WRITE (*,*) "Delete all members!"
      CALL GRID_fDelete (t_gridll)
      i_number_of_members = GRID_fifCount (t_gridll)
      WRITE (*,*) "Number of members in grid linked list:",
i_number_of_members
!
END PROGRAM pTestLinkedList
```

1.1.29 Module

Below we find an example implementation of a module:

```
!+ A linked list wrapper for the derived data type GRID_dfGrid.
!----------------------------------------------------------------------
---------
!! @description
!!  This module is a wrapper module for the module mfGridBase. Ths
module
!!  defines a derived data type extension of GRID_dfGrid. The new
derived
!!  data type contains a character string itdentifying the linked list
item
!!  by a name, a pointer to an instance of the derived type
GRID_dfGrid and
!!  finally a pointer to the next item in the linked list. We refer to
this
!!  derived type as a {.linked list member.}. The linked list is
represented
!!  by the derived data type mfGridBaseLl. This derived data type
consists
!!  one pointer to the head linked list member of the list and one
pointer to
!!  the tail linked list member of the list.
!!
!!  The following public operations or module procedures have been
defined:
```

```
!!
!!{ d
!!  New * Initialize the linked list.
!!  Delete * Delete the entire linked list.
!!  Remove * Remove and deallocate an item from the linked list.
!!  Append * Append item identified by character string to the linked
list.
!!  Copy * Copy one item from the linked list to another item in the
linked
!!         list.
!!  Print * Print a list of all the names identifying the item in the
linked
!!          list.
!!  PointerMember * Get pointer to a linked list member item with a
given
!!                  identifying name.
!!  PointerBase * Get pointer to the base type of a linked list
member.
!!  Count * Count the number of members in the linked list.
!!}
!! We find a list of all the module procedures in the tables below:
!!
!! @procedure_list
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
MODULE mfGridBaseLl
!
! Modules used:
!
!! The base module for the construction of a linked list.
      USE mfGridBase
!
! Impose strong typing of module data and all module procedures:
!
      IMPLICIT NONE
!
! Make the default access of the module private:
!
      PRIVATE
!
! Global (i.e. public) Declarations:
! Global Parameters:
!!
      INTEGER, PARAMETER, PUBLIC :: GRID_ifNAME_LENGTH = 40 !! Maximum
linked
                                                            !! list
member name.
!
! Global Type Definitions:
!
```

```
!! Derived data type definition of a linked list member.
      TYPE, PUBLIC :: GRID_dfGridLlM
         PRIVATE
         CHARACTER (LEN=GRID_ifNAME_LENGTH) :: cName !! Name
associated with linked
                                                     !! list item.
         TYPE (GRID_dfGrid), POINTER :: tGrid !! Instance of
                                              !! derived data type
GRID_dfGrid.
         TYPE (GRID_dfGridLlM), POINTER :: tNext !! Pointer to next
linked list
                                                 !! member item.
      END TYPE GRID_dfGridLlM
!
!! Derived data type definition of the linked list.
      TYPE, PUBLIC :: GRID_dfGridLl
         PRIVATE
         TYPE (GRID_dfGridLlM), POINTER :: tHead !! A pointer to the
head item
                                                 !! of the list.
         TYPE (GRID_dfGridLlM), POINTER :: tTail !! A pointer to the
tail item
                                                 !! of the list.
      END TYPE GRID_dfGridLl
!
! Global Module Subroutines:
      PUBLIC :: GRID_fNew, GRID_fDelete, GRID_fRemove, GRID_fAppend,
GRID_fCopy
      PUBLIC :: GRID_fName, GRID_fPrint
!
! Global Module Functions:
      PUBLIC :: GRID_fifPrint, GRID_ftfPointerMember,
GRID_ftfPointerBase
      PUBLIC :: GRID_fifCount
!
! Local (i.e. private) Declarations:
! Local Scalars:
!! Auxilliary pointers used in the implementation of the module
!! procedures.
      TYPE (GRID_dfGridLlM), POINTER :: tCurrent !! Pointer to the
current
                                                 !! member in the
linked list.
      TYPE (GRID_dfGridLlM), POINTER :: tNext !! Pointer to the next
                                              !! member in the linked
list.
      TYPE (GRID_dfGridLlM), POINTER :: tPrevious !! Pointer to the
previous
                                                  !! member in the
linked list.
!
! Generic functions definitions:
!   Define the generic functions.
!! Definition of the generic {.constructor.}.
      INTERFACE GRID_fNew
         MODULE PROCEDURE GRID_fNewGridLl
         MODULE PROCEDURE GRID_fNewGridLlM
      END INTERFACE
!
```

```
!! Definition of the generic {.destructor.}.
      INTERFACE GRID_fDelete
         MODULE PROCEDURE GRID_fDeleteGridLl
         MODULE PROCEDURE GRID_fDeleteGridLlM
      END INTERFACE
!
!! Definition of the generic {.print operator.}.
      INTERFACE GRID_fPrint
         MODULE PROCEDURE GRID_fPrintGridLl
         MODULE PROCEDURE GRID_fPrintGridLlM
      END INTERFACE
!
CONTAINS
!
! Module operator procedures:
! Define operator procedures contained in this module.
!
! Module procedures:
! Define procedures contained in this module.
!
!+ Initialize the linked list.
!----------------------------------------------------------------------
---------
!! @description
!!  This is the {.base constructor.} for the linked list data type
!!  GRID_dfGridLl. An instance of the derived type is allocated. The
derived
!!  data types components head and tail are nullified.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fNewGridLl (t_This)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(inout):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
!
         INTEGER :: i_status ! Status identifier returned by allocate.
!
         IF ( .NOT. ASSOCIATED(t_This)) THEN
            ALLOCATE (t_This, STAT=i_status)
         END IF
!
         NULLIFY (t_This%tHead, t_This%tTail)
!
      END SUBROUTINE GRID_fNewGridLl
!
!+ Initialize a linked list member.
```

```
!----------------------------------------------------------------------
---------
!! @description
!!  This is a {.constructor.} for the linked list member derived data
type
!!  GRID_dfGridLlM.
!!
!! @method
!!  First we allocate an instance of the derived data type
GRID_dfGridLlM.
!!  The components of the derived type instance is then initialized.
The
!!  name of a linked list member is blanked. The pointer to the base
type
!!  and to the next linked list member item are nullified. We then
apply
!!  the base constructor to the base type.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fNewGridLlM (t_This)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(inout):
         TYPE (GRID_dfGridLlM), POINTER :: t_This !! Pointer to a
linked list
                                                  !! member derived
data type.
!
         INTEGER :: i_status ! Status identifier returned by allocate.
!
!
         IF ( .NOT. ASSOCIATED(t_This)) THEN
            ALLOCATE (t_This, STAT=i_status)
            t_This%cName = " "
            NULLIFY (t_This%tGrid, t_This%tNext)
            CALL GRID_fNew (t_This%tGrid)
         END IF
!
      END SUBROUTINE GRID_fNewGridLlM
!
!+ Delete the entire linked list.
!----------------------------------------------------------------------
---------
!! @description
!!  This is the {.base destructor.} for the linked list.
!!
!! @method
!!  We traverse the linked list and apply the linked list member
destructor
```

```
!!  to each of the members of the list. Finally we deallocate the
instance
!!  of the linked list itself.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fDeleteGridLl (t_This)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(inout):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
!
         INTEGER :: i_status
         TYPE (GRID_dfGridLlM), POINTER :: t_current, t_next
!
         IF ( .NOT. ASSOCIATED(t_This)) RETURN
!
         t_current => t_This%tHead
!
         ListLoop: DO
            IF ( .NOT. ASSOCIATED(t_current)) EXIT ListLoop
!
            t_next => t_current%tNext
            CALL GRID_fDelete (t_current)
            t_current => t_next
         END DO ListLoop
!
         DEALLOCATE (t_This, STAT=i_status)
!
! Cleaning up by nullifying auxilliary pointers.
!
         NULLIFY (t_current, t_next)
!
      END SUBROUTINE GRID_fDeleteGridLl
!
!+ Delete a member of the linked list.
!----------------------------------------------------------------------
---------
!! @description
!!  This is the {.destructor.} for the linked list member derived data
type.
!!
!! @method
!!  We first apply the base type destructor to the base type
component.
!!  Then we deallocate the linked list member instance itself.
!!
!! @owner Magne Rudshaug
```

```
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fDeleteGridLlM (t_This)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(inout):
         TYPE (GRID_dfGridLlM), POINTER :: t_This !! Pointer to the
linked list
                                                  !! data type.
!
         INTEGER :: i_status ! Status identifier returned by
deallocate.
!
         IF ( .NOT. ASSOCIATED(t_This)) RETURN
!
         CALL GRID_fDelete (t_This%tGrid)
!
         DEALLOCATE (t_This, STAT=i_status)
!
      END SUBROUTINE GRID_fDeleteGridLlM
!
!+ Print the entire linked list.
!----------------------------------------------------------------------
---------
!! @description
!!  This is the {.print operator.} for the linked list derived type
!!  GRID_dfGridLl.
!!
!! @method
!!  We traverse the linked list and apply the linked list member print
!!  operator to each of the members of the list
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fPrintGridLl (t_This)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(inout):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
!
```

```
         INTEGER :: i_status ! Status identifier returned by allocate.
!
         tCurrent => t_This%tHead
!
         ListLoop: DO
            IF ( .NOT. ASSOCIATED(tCurrent)) EXIT ListLoop
            CALL GRID_fPrint (tCurrent)
            tCurrent => tCurrent%tNext
         END DO ListLoop
!
! Cleaning up by nullifying module local pointers.
         NULLIFY (tCurrent)
!
      END SUBROUTINE GRID_fPrintGridLl
!
!+ Print a member of the linked list.
!----------------------------------------------------------------------
---------
!! @description
!!  This is the {.print operator.} for the linked list member derived
data type
!!  GRID_dfGridLlM.
!!
!! @method
!!  We print the name of the linked list member and then apply the
print
!!  operator to the instance of the base type.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fPrintGridLlM (t_This)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(inout):
         TYPE (GRID_dfGridLlM), POINTER :: t_This !! Pointer to a
linked list
                                                   !! member derived
data type.
!
!
         IF (ASSOCIATED(t_This)) THEN
            WRITE (*,*) "Name of member : ", t_This%cName
            CALL GRID_fPrint (t_This%tGrid)
         ELSE
            WRITE (*,*) "Not associated !"
            WRITE (*,*)
         END IF
!
      END SUBROUTINE GRID_fPrintGridLlM
!
```

```
!+ Remove and deallocate an item from the linked list.
!----------------------------------------------------------------------
---------
!! @description
!!  This subroutine will {.remove.} (delete) the linked list member
specified
!!  by the linked list member name. If no member by the specified name
exists
!!  nothing will be done.
!!
!! @method
!!  The linkde list is traversed to find the member by the specified
name.
!!  We then applies the linked list member destructor to this member
and
!!  the linked list is joined so that no list member is lost.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fRemove (t_This, c_Name, i_Err)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(in):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
         CHARACTER (LEN=*), INTENT (IN) :: c_Name !! Name identifying
the list
                                                  !! item
!
! Scalar arguments with intent(out):
         INTEGER, INTENT (OUT) :: i_Err !! Error status identifier.
!
         TYPE (GRID_dfGridLlM), POINTER :: t_previous, t_current,
t_next
!
!
         i_Err = 0
!
         IF ( .NOT. ASSOCIATED(t_This)) THEN
   ! Linkde list does not existent
            i_Err = 1
            RETURN
         END IF
!
         t_current => t_This%tHead
!
         IF (TRIM(t_This%tHead%cName) == c_Name) THEN
   ! Remove the linked list head.
            t_This%tHead => t_current%tNext
            CALL GRID_fDelete (t_current)
         ELSE
   ! To delete an item inside the list we need to know the previous
item.
            NULLIFY (tPrevious)
            ListLoop: DO
               IF (ASSOCIATED(t_current%tNext)) THEN
                  IF (TRIM(t_current%tNext%cName) == c_Name) THEN
                     t_previous => t_current
                     EXIT ListLoop
                  ELSE
                     t_next => t_current%tNext
                     t_current => t_next
                  END IF
               END IF
            END DO ListLoop
!
            IF (ASSOCIATED(t_previous)) THEN
      ! Item by this name in linked list
               t_current => t_previous%tNext
               t_next => t_current%tNext
               t_previous%tNext => t_next
               CALL GRID_fDelete (t_current)
            ELSE
      ! No item by this name in list
               i_Err = 2
            END IF
         END IF
!
! Cleaning up by nullifying auxilliary pointers.
!
         NULLIFY (t_previous, t_current, t_next)
!
      END SUBROUTINE GRID_fRemove
!
!+ Append linked list member member identified by member name.
!----------------------------------------------------------------------
---------
!! @description
!!  This subroutine {.appends.} a linked list member of type
GRID_dfGridLlM
!!  to the linked list. The linked list will have the name specified.
!!  of the linked list member will be have to
!!
!! @method
!!  The new linked list member is created by applying the linked list
member
!!  constructor. This means that base type is has been created by the
base
!!  constructor. This means that further initialization must be
provided
!!  to define the linked list member completely.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fAppend (t_This, c_Name, i_Err)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(in):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
         CHARACTER (LEN=*), INTENT (IN) :: c_Name !! Name identifying
the list
                                                  !! member.
!
! Scalar arguments with intent(out):
         INTEGER, INTENT (OUT) :: i_Err !! Error status identifier.
!
         i_Err = 0
!
         IF (ASSOCIATED(t_This%tHead)) THEN
            CALL GRID_fNew (tCurrent)
            tCurrent%cName = c_Name
            tCurrent%tNext => t_This%tHead
            t_This%tHead => tCurrent
         ELSE
            CALL GRID_fNew (tCurrent)
            tCurrent%cName = c_Name
            NULLIFY (tCurrent%tNext)
            t_This%tHead => tCurrent
            t_This%tTail => tCurrent
         END IF
!
! Cleaning up by nullifying auxilliary pointers.
!
         NULLIFY (tCurrent)
!
      END SUBROUTINE GRID_fAppend
!
!+ Copy one linked list member to another linked list member.
!----------------------------------------------------------------------
---------
!! @description
!!  This subroutine {.copies.} a linked list member to another linked
list
!!  member. The action of the subroutine may be summarized as:
!!
!!{ d
!!  {.From.} linked list member exist: * Check if {.to.} linked list
member
!!                                       exist.
!!{ d
!!   {.To.} linked list member exist: * Copy contents of the {.from.}
base type
!!                                      to the {.to.} base type.
Everything
!!                                      else is left unchanged.
!!   {.To.} linked list member doesn't exist: * We append a new linked
list
!!                                              member to the list
with the
!!                                              {.to.} name. We then
copy the
!!                                              contents of the
{.from.} base
!!                                              type to the {.to.}
base type.
!!                                              The {.to.} linked list
member
!!                                              becomes the new head
of
!!                                              the linked list.
!!}
!! {.From.} linked list member doesn't exist: * Nothing is done.
!!}
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fCopy (t_This, c_From, c_To, i_Err)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(in):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
         CHARACTER (LEN=*), INTENT (IN) :: c_From !! Name identifying
the from
                                                  !! list item.
         CHARACTER (LEN=*), INTENT (IN) :: c_To !! Name identifying
the to list
                                                !! member.
!
! Scalar arguments with intent(out):
         INTEGER, INTENT (OUT) :: i_Err !! Error status identifier.
!
! Local scalars:
         TYPE (GRID_dfGridLlM), POINTER :: t_from ! Pointer to the
from linked
                                                  ! list member.
         TYPE (GRID_dfGridLlM), POINTER :: t_to ! Pointer to the to
linked list
                                                ! member.
!
!
         i_Err = 0
         t_from => GRID_ftfPointerMember (t_This, c_From)
!
```

```
         IF (ASSOCIATED(t_from)) THEN
            t_to => GRID_ftfPointerMember (t_This, c_To)
            IF (ASSOCIATED(t_to)) THEN
               t_to%tGrid = t_from%tGrid
            ELSE
               CALL GRID_fNew (t_to)
               t_to%cName = c_To
               t_to%tGrid = t_from%tGrid
               t_to%tNext => t_This%tHead
               t_This%tHead => t_to
            END IF
         ELSE
            i_Err = 1
         END IF
!
! Cleaning up by nullifying auxilliary pointers.
!
         NULLIFY (t_from, t_to)
!
      END SUBROUTINE GRID_fCopy
!
!+ Print a list of the linked list member names.
!----------------------------------------------------------------------
---------
!! @description
!!  This function {.prints.} a list of the linked list member names.
!!
!! @method
!!  Traverse the list and print the name of the linked list member.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      FUNCTION GRID_fifPrint (t_This) RESULT (i_Err)
!
! Declarations:
! Function arguments:
! Scalar arguments with intent(in):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
!
! Scalar arguments with intent(out):
         INTEGER :: i_Err !! Error status identifier.
!
!
         i_Err = 0
         tCurrent => t_This%tHead
!
         ListLoop: DO
            IF ( .NOT. ASSOCIATED(tCurrent)) EXIT ListLoop
!
```

```
            WRITE (*,*) "Name of list member:", tCurrent%cName
            tCurrent => tCurrent%tNext
         END DO ListLoop
!
! Cleaning up by nullifying auxilliary pointers.
!
         NULLIFY (tCurrent)
!
      END FUNCTION GRID_fifPrint
!
!+ Get pointer to a linked list member with a given name.
!----------------------------------------------------------------------
---------
!! @description
!!  This function returns a {.pointer to the linked list member.} by
the
!!  specified name. If no linked list member of the given name exists
!!  the null pointer will be returned.
!!
!! @method
!!   Traverse the list to find a linked list member by the given name.
!!   If one is found a pointer to this linked list member is returned.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      FUNCTION GRID_ftfPointerMember (t_This, c_Name) RESULT
(t_ThisMember)
!
! Declarations:
! Function arguments:
! Scalar arguments with intent(in):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
         CHARACTER (LEN=*) :: c_Name !! Name of linked list member.
!
! Result argument:
         TYPE (GRID_dfGridLlM), POINTER :: t_ThisMember !! Pointer to
the
                                                        !! linked list
member
                                                        !! data type.
!
!
         NULLIFY (t_ThisMember)
!
         IF ( .NOT. ASSOCIATED(t_This)) RETURN
         tCurrent => t_This%tHead
!
         ListLoop: DO
            IF ( .NOT. ASSOCIATED(tCurrent)) EXIT ListLoop
```

```
!
            IF (TRIM(tCurrent%cName) == c_Name) THEN
               t_ThisMember => tCurrent
               EXIT ListLoop
            ELSE
               tCurrent => tCurrent%tNext
            END IF
         END DO ListLoop
!
! Cleaning up by nullifying auxilliary pointers.
!
         NULLIFY (tCurrent)
!
      END FUNCTION GRID_ftfPointerMember
!
!+ Get pointer to the base type of a linked list member with given
name.
!----------------------------------------------------------------------
---------
!!  This function returns a {.pointer to the base type of the linked
list
!!  member.} by the specified name. If no linked list member of the
given name
!!  exists the null pointer will be returned. This function will be
useful
!!  when further initialization of the base type needs to be
performed.
!!
!! @method
!!   Traverse the list to find a linked list member by the given name.
!!   If one is found a pointer to the base type of this this linked
list
!!   member is returned.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      FUNCTION GRID_ftfPointerBase (t_This, c_Name) RESULT
(t_BaseType)
!
! Declarations:
! Function arguments:
! Scalar arguments with intent(in):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
         CHARACTER (LEN=*) :: c_Name !! Name of linked list member.
!
! Result argument:
         TYPE (GRID_dfGrid), POINTER :: t_BaseType !! Pointer to the
base data
                                                   !! type.
```

```
!
!
         NULLIFY (t_BaseType)
!
         IF ( .NOT. ASSOCIATED(t_This)) RETURN
         tCurrent => t_This%tHead
!
         ListLoop: DO
            IF ( .NOT. ASSOCIATED(tCurrent)) EXIT ListLoop
!
            IF (TRIM(tCurrent%cName) == c_Name) THEN
               t_BaseType => tCurrent%tGrid
               EXIT ListLoop
            ELSE
               tCurrent => tCurrent%tNext
            END IF
         END DO ListLoop
!
! Cleaning up by nullifying auxilliary pointers.
!
         NULLIFY (tCurrent)
!
      END FUNCTION GRID_ftfPointerBase
!
!+ Count the number of members in the linked list.
!----------------------------------------------------------------------
---------
!! @description
!!  This function returns the {.number of linked list members in the
linked
!!  list.}.
!!
!! @method
!!  Traverse the linked list and count the number of linked list
members.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      FUNCTION GRID_fifCount (t_This) RESULT (i_NumberOfMembers)
!
! Declarations:
! Function arguments:
! Scalar arguments with intent(in):
         TYPE (GRID_dfGridLl), POINTER :: t_This !! Pointer to the
linked list
                                                 !! data type.
!
! Result argument:
         INTEGER :: i_NumberOfMembers !! The number of members in the
linked
                                      !! list.
```

```
!
!
         i_NumberOfMembers = 0
!
         IF ( .NOT. ASSOCIATED(t_This)) RETURN
         tCurrent => t_This%tHead
!
         ListLoop: DO
            IF ( .NOT. ASSOCIATED(tCurrent)) EXIT ListLoop
!
            i_NumberOfMembers = i_NumberOfMembers + 1
            tCurrent => tCurrent%tNext
!
         END DO ListLoop
!
! Cleaning up by nullifying auxilliary pointers.
!
         NULLIFY (tCurrent)
!
      END FUNCTION GRID_fifCount
!
!+ Returning the name of a linked list member.
!----------------------------------------------------------------------
---------
!! @description
!!  This function returns the {.name of a linked list member.}.
!!
!! @method
!!  Check if the linked list member exists. If it exists return the
name of
!!  linked list member.
!!
!! @owner Magne Rudshaug
!!
!! @history
!! 0.10  02.09.98  Magne Rudshaug  Original code.
!!
!! @language Fortran 90/95
!! @standard Programming standard for IfeFEM 3.0
!----------------------------------------------------------------------
---------
!
      SUBROUTINE GRID_fName (t_This, c_Name, i_Err)
!
! Declarations:
! Subroutine arguments:
! Scalar arguments with intent(in):
         TYPE (GRID_dfGridLlM), POINTER :: t_This !! Pointer to the
linked list
                                         !! member data type.
!
! Scalar arguments with intent(out):
         CHARACTER (LEN=GRID_ifNAME_LENGTH), INTENT (OUT) :: c_Name !!
Name
                                         !! identifying the
list
                                         !! member.
         INTEGER, INTENT (OUT) :: i_Err !! Error status identifier.
!
```

```
!
         i_Err = 0
         c_Name = " "
!
         IF (ASSOCIATED(t_This)) THEN
            c_Name = t_This%cName
         END IF
!
      END SUBROUTINE GRID_fName
!
END MODULE mfGridBaseLl
!
```

Documentation

Documentation may be split into two categories: external documentation , outside the code; and internal documentation , inside the code. These are described in sections 5.1 and 5.2 respectively. In order for the documentation to be useful it needs to be both up to date and readable outside Institute for Enery Technology. To ensure this all documentation, both internal and external, shall be available in English. To enforce this we introduce the convention:

Documentation convention 1 *Documentation shall be provided in English in the formats LATEX2e and HTML. Graphics included in the documentation shall be provided in the formats EPS (encapsulated postscript, LATEX2e) and GIF (graphics interface format, HTML).*

The LATEX2e format is suitable for paper and HTML for online presentation.

A complete Fortran 90/95 program or library (a collection of object files maintained by a librarian) will be referred to as a *package* in the text below.

External

In most cases this will be provided at the package level, rather than for each individual procedure. It shall include the following:

Top Level Scientific documentation:

this defines the problem being solved by the package and the scientific rationale for the solution method adopted. This documentation should be independent of (i.e. not refer to) the code itself.

Implementation documentation:

this documents a particular implementation of the solution method described in the scientific documentation. All program units (subroutines, functions, modules etc...) in the package should be listed by name together with a brief description of what they do. A calling tree for routines within the package must be included.

A User Guide:

this describes in detail all inputs into the package. This includes both procedure arguments to the package and any switches or 'tuneable' variables within the package. Where appropriate default values; sensible value ranges; etc should be given. Any files or namelists read should be described in detail.

Internal

This is to be applied at the individual procedure level. There are four types of internal documentation, all of which shall be present.

Documentation comments:

every program unit must have documentation comments. The purpose of documentation comments is to describe the function of the routine, probably by referring to external documentation, and to document the variables used within the routine. All variables used within a program unit must be declared and commented as to their purpose. It is a requirement of this standard that the templates, defined for each program unit and operator in section 4, be used and completed fully. Extra documentation sections may be added to these headers if a user so wish. With this practice a formatted (LATEX2e and HTML) version of the internal documentation may be generated automatically by the program **f90toDOC** , see section 6.2. The formatted version of the internal documentation will be referred to as the *application programmers interface* or for short simply the *API* . The recommended format for documentation comments is:

```
!-----------------------------------------------------------------
--------

!! <Ordinary text and f90toDOC formatting directives>

!-----------------------------------------------------------------
--------
```

If the documentation comments block is small we use the simplified format:

```
!! <Ordinary text and f90toDOC formatting directives>
```

See section 6.2 for a definition of the **f90toDOC** formatting dirctives.

Section comments:

these divide the code into numbered logical sections and may refer to the external documentation. These comments must be placed on their own lines at the start of the section they are defining. The recommended format for section comments is:

```
!-----------------------------------------------------------------
--------

! <Section number> <Section title>

!-----------------------------------------------------------------
--------
```

where the text in <> is to be replaced appropriately.

General comments:

these are aimed at a programmer reading the code and are intended to simplify the task of understanding what is going on. These comments must be placed either immediately before or on the same line as the code they are commenting. The recommended format for these comments is:

```
! <Comment>
```

where the text in <> is to be replaced appropriately.

Meaningful names:

code is much more readable if meaningful words are used to construct Fortran 90/95 entity names.

API Documentation tools

An important aspect of the documentation of IfeFEM 3.0 is the documentation of the application programers interface (API) . A major problem with such documentation is to keep it up to date. A preferred way to accommodate this is to keep the documentation of the API in the source code as header comments. We then need a tool to format these header comments and other relevant information to produce the actual API documentation. Since no suitable tool existed we have developed our own tool **f90toDOC** for this purpose. The formats supported by **f90toDOC** are $\LaTeX$2e and HTML . The availability of the source code itself is also very important for the purpose of documentation. To simplify the navigation through source files we translate it to the HTML format where all definitions and inclusions are resolved through hyper links. We have developed the tool **f90toHTML** for this purpose.

F90toHTML

One of the main reasons for using the HTML format is to take full advantage of hyper links. To resolve all possible links in a group of inter related Fortran 90/95 source files we need to know all definitions in the source files in advance. Thus, before we can convert the source files to the HTML format we have to make an analysis with respect to definitions in all the source files involved. The result of such an analysis is a *fortran definition file* (fdb file) . We will use the extension `.fdb` to denote such files. When all the files have been analyzed and a fdb file has been produced, we may start converting source files to the HTML format. In the conversion process references will be resolved through the definitions in the fdb file.

1.1.30 Usage

f90toHTML [-b:FDBfile_out] [-B:FDBfile_in] [-raw] `file`

-b:`FDBfile_out`

With this option the program performs an analysis of the Fortran 90/95 definitions in the input file. The definitions detected are stored in the file `FDBfile_out`. The -B option should not be specified in this mode. The default fdb output file is `f90toHTML.fdb`.

-B:`FDBfile_in`

With this option the program performs a HTML conversion of the input file. Unknown references are resolved through the fdb file `FDBfile_in` and must be supplied. The -b option should not be specified in this mode. The default fdb input file is `f90toHTML.fdb`.

-raw

Raw HTML output (omit header and tail) for inclusion in other files.

`file`

The Fortran 90/95 input file.

In the HTML conversion mode output is directed to standard output. Example of analysis usage:

```
f90toHTML -b:IfeFEM.fdb mfModule.f90
```

In this example the definitions of the `mfModule.f90` source file has been appended to the fdb file `IfeFEM.fdb`. If new definitions are added to `mfModule.f90` you run **f90toHTML** in the same way and the fdb file will be updated. To produce a complete fdb file, **f90toHTML** must be run in the same manner for all inter related Fortran 90/95 source files. When this is done we are ready to start conversion. Example of conversion usage:

```
f90toHTML -B:IfeFEM.fdb mfModule.f90 > mfModule.html
```

In this example the Fortran 90/95 file `mfModule.f90` will be converted to HTML format. The references will be resolved through the fdb file `IfeFEM.fdb`. The output from the program has been redirected to the file `mfModule.html`.

1.1.31 Example

As an example of the use of **f90toHTML** we converted the source file, `pTestLinkedList.f90` in the program example in section 4.3.1 and `mfGridBaseLl.f90` in the module example in section 4.3.2 to HTML format. To complete the example we also need the files `mfGridBase.f90` and `mfGrid.f90`.

F90toDOC

f90toDOC is a program written to produce API documentation from Fortran 90/95 source code. It is based on special source code comments. The appearance of a comment block signals that the next statement with a recognizable Fortran 90/95 keyword shall be documented. The format of the documentation produced by **f90toDOC** is $\LaTeX$2e or HTML.

Usage of **f90toDOC** assumes that an analysis with **f90toHTML** of the files to be documented has been performed. The resulting fdb file from the analysis is used in the documentation process.

1.1.32 Source code comments

The **f90toDOC** comment symbol was chosen with the constraint that it must be interpreted as a Fortran 90/95 comment, it should not clutter the source to much and must be different than the Fortran 90/95 comment sign `!`. The last constraint allows some part of the code to be commented in the Fortran 90/95 sense, but not in the **f90toDOC** sense. We chose `!!` as the **f90toDOC** comment symbol. You can place a block of comments before the statement to be commented. A `!!` type comment, starting in column one, before a statement signals to **f90toDOC** that this statement shall be documented. For example:

```
!! Support for error handling
MODULE mfError
```

When declaring a variable, a comment is used to define the meaning of the variable. In this case it is natural to append the comment to the statement as shown in the following example:

```
REAL, TARGET :: RealWork(:) !! Real working array
```

This in itself will not suffice for a documentation of the variable `RealWork`. This type of comment should be regarded as an attribute to the declaration, and will be used as a part of the documentation if the declaration is preceded by an ordinary **f90toDOC** comment as shown in the example below.

```
!! Intrinsic type work arrays.
REAL, TARGET :: RealWork(:)       !! Real working array
INTEGER, TARGET :: IntegerWork(:) !! Integer working array
```

COMPLEX, TARGET :: ComplexWork(:) !! Complex working array

1.1.33 Formatting in f90toDOC comments

To write API documentation we need a minimum of formatting possibilities. The formatting possibilities in **f90toDOC** are restricted to simple *lists* , *type-face modifiers* and *macros* .

Lists are started with the modifier `!!{ [u,o,d]` and ended with the modifier `!!}`. The starting modifier has an optional argument expressing what type of list it is. `u` gives an unordered list (same as LaTeX2e itemize), the default. `o` gives an ordered list (same as LaTeX2e enumerate) and `d` gives a definition list (similar to LaTeX2e description). The items of the list has the general format:

```
    entity * item text
```

Here `entity` is only relevant in a definition list. Here it is the entity to be defined by the `item text`.

The *type-face modifiers* are defined with a start and end modifier. The modifiers may be of type in-line or multi-line. The in-line modifiers are **bold**, *emphasize* and `verbatim`. The only multi-line modifier implemented so far is `verbatim`. The modifiers are shown in the following example:

```
    !! We may have [.bold.] text, <.verbatim.> text and
    !! {.emphasized.} text in-line in comments. We may also have
    !! a multi-line verbatim text.
    !!<
    !! This is line 1 of verbatim text.
    !! This is line 2 of verbatim text.
    !!>
```

Nine *macros* are currently supported. A macro has the general format:

```
    @macro-name macro-argument
```

The macros are:

description

Produce a header for the description part of the documentation.

method

Produce a header for the method part of the documentation.

owner

The macro takes one argument, the name of the person presently responsible for the maintenance of the program unit in question.

history

The macro takes the revision history as input. The revision history must start on the line immediately following the macro. One line in the revision history consists of four items. The revision identifier, the date of the revision, the signature of person responsible for revision and the comment describing what has been done in the revision. The revision items must be separated by at least two blank spaces. Revision items may be continued over several lines. The only requirement being that the items must start in the same column.

input_files

Produce a header for the decription of the use of input files in the program unit.

output_files

Produce a header for the decription of the use of output files in the program unit.

language

The macro takes one argument, the programming language used in the implementation.

standard

The macro takes one argument, the reference to the programing standard used.

procedure_list

The macro generates a list of the subroutines and functions defined on the current file. The list is positioned immediately after the macro. The macro takes no arguments.

see

The macro takes one argument, a comma separated list of "See also" references to other related program entities or units.

index

The macro takes one argument, a parameter defining what type of index entry shall be generated for the entity in the comment block. With the argument "main" the commented block is made the main index in the LATEX2e index list. This macro has no meaning in the case of HTML documentation.

header

The macro takes one argument, the header title to be used instead of the header defined automatically. Each comment block automatically gets a header determined by what entity is being commented. This macro lets you override this choice of header.

append_header

The macro takes one argument, a text describing the header text more detailed. If the header for some reason needs a more descriptive text this may be appended with this macro.

name

The macro takes one argument, the name of the entity being documented. With this macro you may overrule the name of the entity being commented.

append_name

The macro takes one argument, a text describing the name more fully. If the name for some reason needs a more descriptive text this may be appended with this macro.

Error condition handling

In the IfeFEM 3.0 project we attempt to introduce a unified error handling system. This will consist of a Fortran 90/95 module mfError containing *global error messages* and global procedures for printing error messages. Since the error messages in this module are general, they must be declared public in the module. In this way they will be available to all other modules using the module mfError. For the mfError we introduce the convention:

Naming convention 23 *`Module acronym`: The module acronym for the mfError module shall be `ERRMH`.*

We introduce the following naming convention for global error messages:

Naming convention 24 *`Global error message`: Shall have the prefix `ERRMH_ef`. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

Error messages corresponding to error conditions that aren't general will be referred to as *local error messages* or *module error messages* . Since this type of error messages will be specific to the module, we declare them as private relative to the module. We introduce the following naming convention for module error messages.

Naming convention 25 *`Local error message`: Shall have the prefix `e`. The components shall start with a capital letter and for the remaining part have lower case letters and numerals and no separators.*

We also introduce a convention for the declaration of error messages.

Syntax convention 13 *IfeFEM 3.0 error messages shall be declared in the following way:*

```
   CHARACTER(LEN=*), PARAMETER :: "error_message"
```

Here `error_message` is any type of error message.

More details on the definition of the IfeFEM 3.0 error handling system will be supplied later.

The IfeFEM 3.0 Directory Structure

The IfeFEM 3.0 project will contain a large number of files. A structuring of these files will certainly simplify the maintenance. It will also be much easier to reference files in the system if a logical directory structure is devised. It is the objective of this section to define a standard *IfeFEM 3.0 Directory Structure* (IDS) : a directory hierarchy for IfeFEM 3.0 source files, makefiles, examples, documentation, tools, and more.

The root directory of the IDS shall be `IfeFEM`. The top level directories of the IDS are:

`source`

for the source files of IfeFEM 3.0 modules.

`lib`

for compiled versions of the IfeFEM 3.0 library.

`standards`

for standards used in the implementation of IfeFEM 3.0.

`tools`

for necessary tools in the IfeFEM 3.0 project.

1.1.34 8.1 The `source` directory

This directory shall contain the subdirectories:

`applications`

for simple IfeFEM 3.0 applications useful for debugging purposes. There will be one directory, containing a makefile and files with extension `.f90`, for each application.

`templates`

for the program unit templates defined in section 4 and templates for implementation of generic algorithms. There will be one file with the extension `.f90` for each template.

`makefiles`

for IfeFEM 3.0 related makefiles. This directory will have a subdirectory for each supported platform. In the subdirectory there will be a makefile building the IfeFEM 3.0 library for the particular platform.

`latex`

for $\LaTeX$2e macros defined and used in the IfeFEM 3.0 project.

⟨*module*⟩

for the source of IfeFEM 3.0 modules. Each directory shall contain one file defining the module. If the name of the module is mfModuleName the directory name shall be `mfModuleName`, and the file name shall be `mfModuleName.f90`. For each module procedure, fProcedureName in module mfModuleName, there shall be a file `fProcedureName.f90` containing the source. HTML versions of the source files will be produced by **f90toHTML**. These files will have the same name, but have the extension `.html`. The API documentation produced by **f90toDOC** will be stored in a subdirectory `doc`. They will have the same name, but extensions `.tex` and `.html`.

The `lib` directory

This directory will have a subdirectory for each supported platform. The subdirectories will contain compiled versions of the IfeFEM 3.0 library for the supported platforms respectively.

The `standards` directory

This directory will contain the standardization documents relevant for the IfeFEM 3.0 project. Standardization documents shall be provided in the $\LaTeX$2e and HTML formats if possible. There will be subdirectories `latex` and `html` allowing separate storage.

The `tools` directory

This directory shall contain the subdirectories:

`bin`

for executable versions of the supported tools. These will be stored in a subdirectory for each of the supported platforms.

`source`

for the source of the supported tools if available. There should be one directory for each tool. This should have a subdirectory `doc` containing available documentation.

Programming, conversion and documentation tools

In this section we will briefly discuss some tools that we find useful in the standardization of IfeFEM 3.0.

F90ppr

This is a Fortran 90/95 preprocessor and source code formatter. The preprocessor is very useful for conditional compilation purposes. Source code formatting may be useful to produce a source code with a standardized layout.

Gnu make

The make utility is very valuable when working with a large set of files. With make a number of tasks may be automated. The problem with make is that the makefile syntax is not standardized. However, the gnu make is supported on all major platforms including win32. Thus, by designing our makefiles for gnu make, we will achieve a much greater degree of standardization.

TeX2HTML

The typesetting system used to produce the printed IfeFEM 3.0 documentation will be $\LaTeX$2e. To make the documentation available electronically it should be converted to a HTML document. TeX2HTML (commercial version of tth) provides this service. This product has several benefits compared to similar products. The program is very simple to install and converts very fast. It deviates from similar products in that it produces mathematics in HTML code, not as pictures. This makes the resulting HTML code containing mathematics to appear much faster on the web browser.

Listings and lgrid

There exists two packages to accommodate inclusion of Fortran 90/95 source code in a $\LaTeX$2e document. These are **listings** and **lgrind**. Both have Fortran 90/95 support. The most visually pleasing result is obtained with **lgrind**. However, **lgrind** involves a separate conversion program. With **listings** source code can be included directly.

Quality assurance

Intentionally left blank.

References

[1]

Michael Metcalf and John Reid. *Fortran 90/95 explained*. Oxford University Press, 1996.

Index (showing section)