MOHID FRAMEWORK PROGRAMMERS MANUAL

Introduction

Objectives

This report aims to serve as a guide to MOHID Framework programmers. It's not meant to contain all the information needed to program in MOHID Framework, but rather serve as guidance for some technical aspects that the programmer must keep in mind in order to write and edit MOHID Framework source code.

General overview

MOHID Framework is the name called to the source code structured organization, designed to create scientific software applications, by the MARETEC team, at Instituto Superior Técnico, Technical University of Lisbon, Portugal. This framework is written, with some exceptions, in ANSI FORTRAN 95. Source code written according to the F95 standards assures platform independency, meaning MOHID Framework applications can run in any operative system that supports a F95 compiler.

Each MOHID Framework project is designed in a modular way, each MODULE corresponding to class. The classes that form MOHID Framework were designed on a common basis, regarding programming rules and definition concepts in order to establish a straightforward connection of the whole code. This is reflected in memory organization, public methods systematisation, possible object states, client/server relations and errors management (Leitão, 2003, Braunschweig, 2004).

Each class is responsible for managing a specific kind of information. The design of a class, in FORTRAN 95, can be accomplished by the MODULE statement. This way, information can be encapsulated using the PRIVATE statement. Encapsulation assures that all the information associated to an object is only changed by the object itself, reducing errors due to careless information handling in other classes.

Source Safe Database

In a software project like MOHID Framework, the number of programmers is both large and variable, turning source code management a primarily task. This effort must be made in order to maintain updated and reliable an extensive source code, containing more than 250000 code lines.

In addition, all source code is kept under a data base project, provided by a specific software[1], which allows centralizing the source code files and keeping multiple versions of each file, as well as document all the changes performed. It performs graphical comparisons of different versions of each file and manages the user access to the code and prevents that more than one user changes the code in one file at the same time (Braunschweig, 2001). The possibility to access the historical record of the code has proven to be an important feature to improve the code robustness as it leads to a fast and reliable error detection.

Important rules

Code alignment

Source code must be as much as possible well aligned, in order to maintain an easy readability for all developers. Here is a guideline to align Mohid source code in a standard way.

1) Be sure to have in your source code editor the definitions to convert a TAB to 4 empty spaces. All TABS in source code must be erased.

2) Exception made to the MODULE and END MODULE statements code lines shall begin at column 5 (that corresponds to one TAB spacing). Code lines starting at column 5, shall include SUBROUTINE or END SUBROUTINE statements.

3) Code inside a subroutine should be placed one TAB inside the SUBROUTINE and END SUBROUTINE statements (column 9).

4) Variables declaration must be aligned by the type of variable, the dimensions, the intent statement or the pointer attributes. The "::" delimiter must also be aligned for all variables.

[1] Microsoft Visual Source Safe Explorer 6.0

5) When using IF…ELSE…END IF statements, this shall be aligned in the same column. All code inside this structure should be placed one TAB inside. If this structure is already inside another IF…ELSE…END IF, the same rule is to be applied.

6) DO…ENDDO structures shall have the same rule as in point 4), but with the exception made to chained DO loops, that shall be aligned in the same column.

7) The one TAB inside rule must also be applied to similar structures like TYPE…END TYPE, SELECT CASE…END SELECT, INTERFACE…END INTERFACE, and so on.

Code declarations

1) When declaring the USE statement, at the beginning of a module, be sure that all the declarations are really used, as it optimize the compiling speed and memory usage.

2) Always use the ONLY statement when using the USE statement, to optimize modules linkage.

3) When declaring variables within a derived TYPE, be sure to erase variables that are not used, as they are not detected as unused variables by the compiler.

4) Be sure to erase all unused or obsolete source code (commented or not commented).

5) Declare all programmed subroutines at the beginning of each module and align them (placing a tab) in the form of a hierarchical tree.

6) When naming IF structures, avoid names using IF (e.g. if1, if2, …), some compilers may have some problems dealing with that syntax;

7) Define error labels, placing the name of the subroutine, the name of the module and an error ID (e.g. ConstructModel – ModuleModel – ERR10); It is recommended to number the error ID, using a 10 number interval, in order to add new errors without having to renumber all the error messages above. Be sure not to have equal errors messages, otherwise the model can stop in one of them and the user will be confused.

8) One of the most used subroutine in Mohid source code is subroutine GetData, used to access information from an input data file, given by a certain keyword. Each time

the subroutine is used, the search for the keyword value is logged in the UsedKeywords file. In order to properly log all used keywords, be sure to include in the call to GetData subroutine, the ClientModule argument, that must indicate the module from where the call is made. Also, when applicable, use the Default argument, which allows the definition of a default value for that keyword, in case this keyword is not found in the input data file.

Creating a new module

In order to create a new module and include it in Mohid Framework, one should always use ModuleShell as the starting file. Exceptions can me made if it is proven that starting from another module is more profitable. In this case, keep in mind that all source code actualizations must be made to the 2 modules (old and new) in order not to spread "dirty" code to new modules.

ModuleShell is the shell of a standard Mohid module, consisting of an almost empty module, which contains only module management code, used to fulfill the object oriented philosophy in which Mohid Framework is based.

Remember to include, if applicable, the module keywords at the source code file header, as it is made in other Mohid modules.

Creating a new subroutine

In order to create a new subroutine in a module, be sure to maintain the alignment rules described above as well as declare it at the top of the MODULE.

In great majority of subroutines, variables are passed as arguments; variables are accessed from other modules and variables are local to that subroutine. Thus, one shall organize the subroutine variables declaration zone in the following way:

1) !Arguments------------------------------Below this commented line should be declared the variables passed as arguments, by the order they are defined in the argument list;

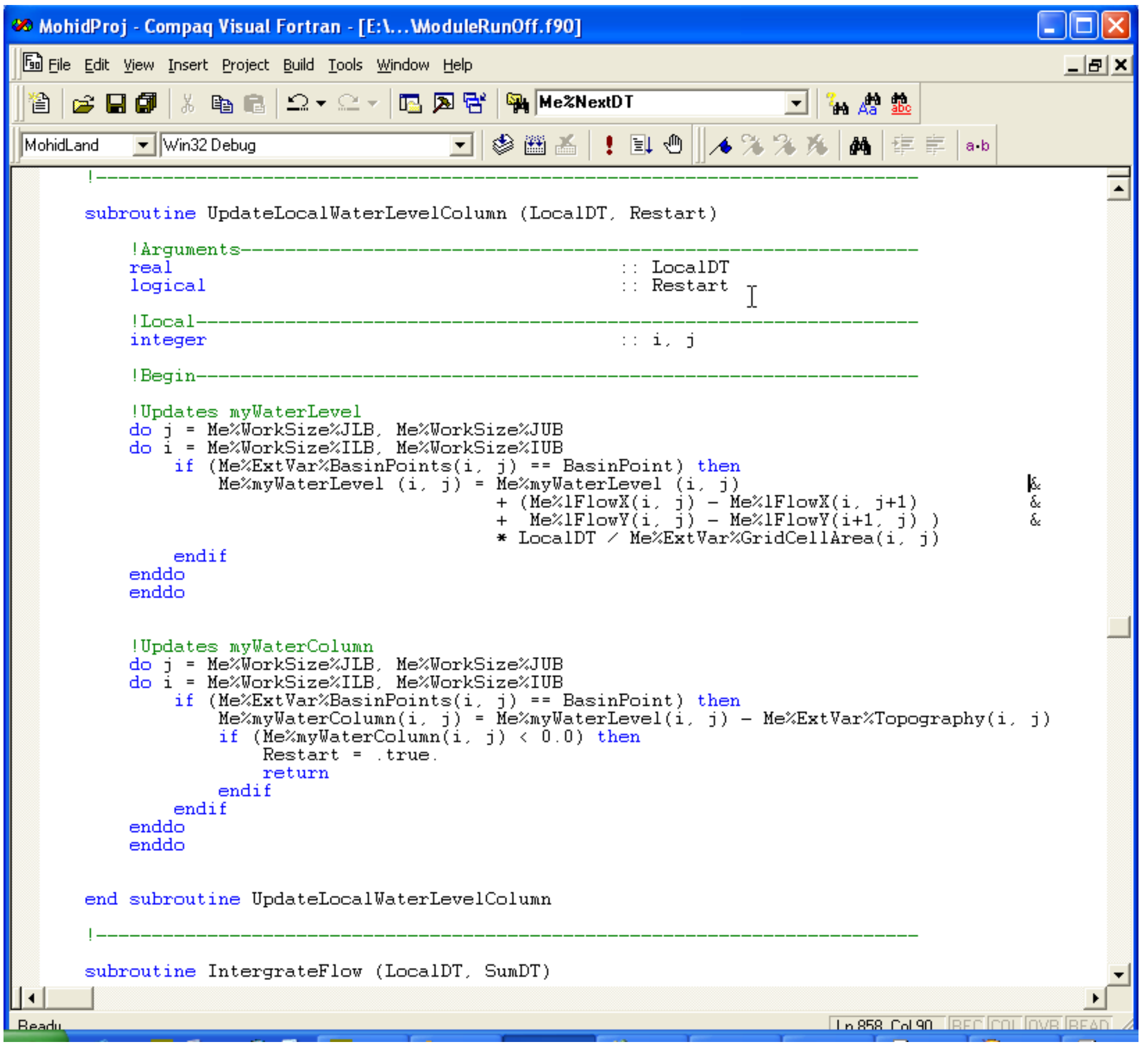
2) !Local-----------------------------------Below this line variables which are only used inside the subroutine should be declared;

3) !Begin-----------------------------------Below this line (plus an empty line) effective source code should begin to be written;

Between 2 subroutines there should be a separating line as below finishing in column 80 (preceded and followed by one empty line):

!---

Figure 1 shows an example of proper formatted subroutine.


MohidProj - Compaq Visual Fortran - [E:\...\ModuleRunOff.f90]
File Edit View Insert Project Build Tools Window Help
Me%NextDT
MohidLand
Win32 Debug
Ready
Ln 858, Col 90

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

    subroutine UpdateLocalWaterLevelColumn (LocalDT, Restart)

        !Arguments-------------------------------------------------------------
        real                                        :: LocalDT
        logical                                     :: Restart

        !Local-----------------------------------------------------------------
        integer                                     :: i, j

        !Begin-----------------------------------------------------------------

        !Updates myWaterLevel
        do j = Me%WorkSize%JLB, Me%WorkSize%JUB
        do i = Me%WorkSize%ILB, Me%WorkSize%IUB
            if (Me%ExtVar%BasinPoints(i, j) == BasinPoint) then
                Me%myWaterLevel (i, j) = Me%myWaterLevel (i, j)                          &
                                         + (Me%lFlowX(i, j) - Me%lFlowX(i, j+1)          &
                                         +  Me%lFlowY(i, j) - Me%lFlowY(i+1, j) )        &
                                         * LocalDT / Me%ExtVar%GridCellArea(i, j)
            endif
        enddo
        enddo


        !Updates myWaterColumn
        do j = Me%WorkSize%JLB, Me%WorkSize%JUB
        do i = Me%WorkSize%ILB, Me%WorkSize%IUB
            if (Me%ExtVar%BasinPoints(i, j) == BasinPoint) then
                Me%myWaterColumn(i, j) = Me%myWaterLevel(i, j) - Me%ExtVar%Topography(i, j)
                if (Me%myWaterColumn(i, j) < 0.0) then
                    Restart = .true.
                    return
                endif
            endif
        enddo
        enddo


    end subroutine UpdateLocalWaterLevelColumn

    !---------------------------------------------------------------------------

    subroutine IntergrateFlow (LocalDT, SumDT)
```

Figure 1: Correctly formatted subroutine

Changing Code

The source code of MOHID is stored in a data base located in the Server *Einstein*. If you are about to change code from MOHID always:

- keep in mind that **you are not the only one who works with MOHID**;

- don’t keep a file checked-out for a long time;
- keep in mind that the model should be easily usable;

Check list before making any changes

- Get the latest version of all the source files;
- Check-out the files which you are going to change;
- Proceed with the changes of the source code; Note that if you want to change a file and it is checked out you cannot proceed with the changes;

Check list before checking in a module

If you have made your changes in a source code file and believe they are finished, walk through this check list before checking in the source code file(s) you have edited.

- Get the latest version of all files from source safe except the ones that are checked out by you. Note that if a module is checked out by another user you must also get the latest version;
- Compile all the code (in single precision and double precision); Verify that code compiles with zero warnings and zero errors.
- Always keep sure that the changes in the code will not affect input and output data files, commonly used options and model configurations. If that is the case, please discuss the changes with a colleague before you check-in the code. After the check-in you are responsible for publicly communicate the changes to the entire MOHID community (programmers and users). If you have doubts that the changes you’ve made are included in this category, please discuss the changes with a colleague.
- If you had or remove any keyword, block, module or program for MOHID Framework, make sure you actualize the keyword database, before you check-in the source code.

Optimizing speed

Loops

- Avoid whenever possible the introduction of decision structures like IF or SELECT CASE inside a DO loop, except when using mapping variables (e.g. Me%WaterPoints3D(i,j,k) == 1).
- Always use (and please change if you step across it) the variable GridCellArea instead of multiplying DUX by DVY. DUX and DVY are constant during the whole simulation so you avoid a multiplication each time step if you use GridCellArea.
- Avoid CALL subroutines inside a 2D or 3D loop. It's very time consuming.
- Whenever possible place the do-loops with the k as outer loop and i as inner loop (for the case that a variable is allocated with (i, j, k)).

Mohid Water (3D loop in the water column)

- Be aware of kFloorZ in order to loop in 3D. The number of layers in each column might not be the same as CARTESIAN discretization can be used. Thus, you must be aware that you must use kFloorZ, the variable that stores the index of the bottom layer, in order to define the limits of your 3D matrixes in the Z coordinate.

Other considerations

Using real numbers without a decimal place

Using real numbers without a decimal place make sure to add a point after the number, otherwise it will be considered as an integer, therefore creating significant rounding errors in multiplication/division operations.

```
Example (OK)
  Mean = (a+b) / 2.
Example (WRONG)
  Mean = (a+b) / 2
```