The code builder

1 Objectives

Monday 7 June 2010
by FJ

A muti-platform tool able to build up the libraries and the executable programs of a software written in C, C++ and/or FORTRAN.

The main idea consists in simplifying the task of a developer when compiling and linking his program. The usual tool for that is "make", possibly associated to "automake" and "autoconf", but its use is not obvious as far as FORTRAN is concerned. Indeed, the module files are not easy to manage and tools like "sfmakedepend" are often applied to generate dependencies.

But "make" has other drawbacks :

- many versions of make exist which are not compatible together, except for the simplest use. Moreover, if constructing a set of makefiles working on a unique platform is relatively easy, managing several development platforms with various operating systems is much more complicated.
- tests on files are applied on the time stamp only. This is a problem with FORTRAN modules, when a change of the source file does not really modify the generated module. The module time stamp is updated (except with few compilers) which often implies a cascade of useless compilations. When a compiler does not change the module time stamp, then the "make" program is itself in trouble if this module file has been defined as a target depending on the source file.

So the new proposed tool has the following objectives :

- Using together the file date and a file signature to determine whether or not a file has been modified since the last build. This function needs to save the current build state in a specific file, in the build directory.
- Recompiling a source file also when compiling options have been modified.
- Making possible to build the software several times with several compilers or operating systems, within a particular build directory for each case, all parameters being managed in a single configuration file which replaces the usual makefile(s). This function is very important because most of FORTRAN developers use several compiler suites.
- Managing correctly FORTRAN modules as well as include files.
- Introducing global tasks which make useless to check cross file dependencies when these files belong to or are generated by other tasks : a task is associated to its own set of files and then its own internal dependency list. The developer supplies the cross file dependency check in defining himself the dependencies between the tasks.
- Handling file suffixes and file names in an operating system independent way, even if it will remain possible to test the current operating system and to select specific actions for it.
- Platform options like file suffixes, compiler flags, OS names... are managed by small configuration blocks within the single configuration file.
- All source files are referenced relatively to the root directory of the software. On the

contrary, produced files are referenced relatively to the building directory.

- Parallel compilation of source files, which are not related together, is managed automatically. This parallel processing occurs only within a task, the tasks being executed sequentially.

2 Installing and using the builder

Wednesday 9 June 2010
by FJ

Windows

```
%builder% [-h] [-v] [-d project-directory] [-f configuration-file] [-p pname pvalue] [platform-name]
```

Linux

```
builder [-h] [-v] [-d project-directory] [-f configuration-file] [-p pname pvalue] [platform-name]
```

-h	**short user manual**
-v	**verbose mode : the builder explains why it compiles source files**
-d	**to precise the root directory of the project. By default, this is the current directory**
-f	**to provide the name of the configuration file. By default, the name is *builder.cfg* and that file is located in the root directory of the project**
-p	**passing a parameter (name and value) which can be used in the configuration file**

Windows

The builder program is presently delivered under a self installing executable program : **builder-win32.exe**

It should work on most of Windows versions : 95-98-2000-NT-XP-VISTA

During the installation, two environment variables are defined :

- the variable **odessa** contains the root directory of the installation,
- the variable **builder** contains the command to run.

The builder must be used in a command windows usually created by the program **cmd.exe**. The builder command must than be written **%builder%** because **builder** is an environment variable.

Linux

The Linux version is installed in deflating the archive file in any directory :

```
cd /../test
tar zxvf builder.tar.gz
```

It is advised to define an alias in the shell configuration file. For instance in .bashrc :

```
alias builder="/../test/odessa/proc/builder"
```

3 Configuration file

Monday 7 June 2010
by FJ

Function

The configuration file describes the project and how to build up the associated libraries and executable programs. It replaces the usual makefile(s). Rather than several files (one makefile by source directory for instance), I have chosen to centralize the whole project description in a single file. In fact, the configuration file may be split into several files, a subject which will be explained later on.

This file is written in the ODESSA reader format, even if an equivalent XML form is possible. The ODESSA format is easier to write than XML:

- it is written in free format and the order of data does not matter,
- it is mainly composed of couples (name,value),
- the data are organized in rubrics, a rubric starting by the keyword STRUCTURE and finishing by the keyword END

Be careful about the string management of ODESSA :

- a name is a short string (up to 8 characters), starting by a letter and composed of letters or digits (special characters $ # _ are also considered as letters),
- a short string containing non alphanumerical characters must be surrounded by apostrophes. For instance '.f90' is a correct short string representing a usual FORTRAN file suffix.
- a long string (more than 8 characters) must be delimited by quotation marks. For instance "-O3 -openmp" is a correct value for compiling options.

Let us also notice that a comment starts by the character ! like in FORTRAN.

Before executing the tasks described in that file, the builder checks carefully the configuration file and stops immediately if an anomaly is detected.

The configuration file is usually saved in the root directory of the project. All file names are relative to the root directory of the project. Its standard name is "builder.cfg" but any file name is possible thanks to the option -f followed by the right configuration file name.

The configuration file contains two mains types of rubrics :

- PLATFORM : set of options for a specific installation platform merging together the operating system and the compiler suite. The options are for instance the compiler names, the compilation flags... A platform block has a name, this name being mention as argument to the build command. So a unique PLATFORM block is always selected when running the code builder. If the name is missing, then the first platform is automatically selected.

- TASK : particular task possibly grouping together sub-tasks. A task may have task dependencies. If a task contains source files, then a kind of makefile is constructed automatically, involving local sources files and all generated files (objects and/or modules).

A third type of rubric is also possible to manage exceptions. The name is EXCEPT. An except rubric enables to choose specific compiling options for a particular source file on a particular platform.

Simple example

Let us consider a project associated to a single source directory containing a mixing of FORTRAN and C source files. A single task is enough to generate the executable program :

```
STRUCTURE TASK
  NAME    mycode                ! task name
  PROGRAM mycode                ! main target : the executable program
  DIR     src                   ! the directory containing sources
  SUFFIX  '.f'                  ! the suffixes of the files to compile
  SUFFIX  '.c'
  SUFFIX  '.f90'
END

STRUCTURE PLATFORM              ! description of a first platform
  NAME "unix-intel"
  BUILD "bin/unix-intel"
  OBJ '.o' MOD '.mod'
  LIB '.a' EXE '' DYN '.so'
  STRUCTURE COMPILER            ! compiler description
    SUFFIX '.f90' SUFFIX '.f'   ! source file suffixes
    COMMAND "ifort"             ! command
    FLAGS   "-O3 -openmp"       ! compilation flags
    NOLINK  "-c"                ! the special flags preventing the link phase
    OUT     "-o"                ! the flag to precise the name of the main output file
    INC     "-I"                ! the flag to precise directories for includes and modules
    MOD     "-module"           ! the flag to precise the directory receiving modules
  END
  STRUCTURE COMPILER            ! compiler for C files
    SUFFIX '.c'   SUFFIX '.cc'  ! source file suffixes
    COMMAND "icc"               ! command
    FLAGS   "-O3"               ! compilation flags
    NOLINK  "-c"                ! the special flag preventing the link phase
    OUT     "-o"                ! the flag to precise the name of the main output file
  END
  STRUCTURE PROGRAM             ! the linker
    COMMAND "ifort"             ! command to use.
    OUT     "-o"                ! the flag to precise the program file
    FLAGS   "-openmp"           ! optimizations flags
  END
END
```

In this example, only one platform is defined. This one describes two compilers and sets up their main parameters.

A single task is foreseen : creating the executable program (file name : bin/linux-intel/builder). This task involves to compile first the source files of the directory src. Two types of source files are selected : FORTRAN and C.

4 TASK

Tuesday 8 June 2010
by FJ

```
     *-> TASK
        [ -> BUILD    t        ! optional build path, relative to the root build
directory
        [*-> COND     s        ! conditional block
            *-> IF    i0       ! condition
             -> ...            ! any information of the TASK block
        [*-> DIR      t        ! directory path to locate source files
        [*-> DEPEND   c0       ! possible TASK dependency (name of another task)
        [*-> DEPFILE  c0/t     ! external file dependency
```

```
        [*-> DIRDEP   c0/t     ! directory where looking for dependencies
        [*-> DIROBJ   c0/t     ! directory where looking for object files
        [*-> INST     t        ! instructions to execute
        [*-> FILENAME t        ! specific files to compile
        [*-> LIBRARY  c0/t     ! name of a library needed to make the program
        [*-> DYNLIB   c0/t     ! name of a dynamic library needed to make the program
        [ -> LIBNAME  c0/t     ! name of the library to create
        [ -> DYNNAME  c0/t     ! name of the dynamic library to create
          -> NAME     c0/t     ! the name of the task
        [ -> PROGRAM  c0/t     ! name of the executable program
        [*-> SUFFIX   c0/t     ! suffixes of source files to compile
        [*-> TARGET   t        ! target file associated to instructions
        [*-> TASK     s        ! sub task executed before compiling task source files
             -> ...
```

Several rules apply :

- If a task contains in the same time sources files and sub-tasks, the sub-tasks are executed first.
- Data DIR and SUFFIX are connected together.
- Data PROGRAM and LIBRARY are connected together. More precisely, a datum LIBRARY is only used when creating an executable program.
- Data TARGET and INST are associated together.

How to compile source files

```
STRUCTURE TASK NAME compile
  DIR "source/tool1" SUFFIX '.f90' SUFFIX '.c'
  BUILD "obj"
END
```

All the source files of the directory "source/tool1" with a suffix .f90 of .c will be compiled. The object files will be stored in the sub-directory "obj" of the building directory of the chosen platform.

It is also possible to compile several files giving their name :

```
STRUCTURE TASK NAME compile2
  FILENAME "source/tool1/f1.f90"
  FILENAME "source/tool2/bis.f90"
  FILENAME "source/tool3/foo.c"
  BUILD "obj"
END
```

It is also possible to mix the two examples. The final list of source files is composed of the files found using (DIR,SUFFIX) plus the files (FILENAME).

If a file is mentioned twice, then only one version is retained.

Cross dependencies between all these source files are computed automatically.

Building a library

```
STRUCTURE TASK NAME lib LIBNAME mylib
  DIR "source/tool1" SUFFIX '.c' SUFFIX '.f90'
  FILENAME "source/tool2/f1.f90"
  BUILD "obj"
END
```

This is exactly like a task compiling files, in inserting in addition the keyword LIBNAME followed by the library name. It is possible to create a dynamic library in the same way, in replacing LIBNAME by DYNNAME.

A dynamic library is called a DLL on Windows and a SO (Shared Object) on Unix

Building an executable program

```
STRUCTURE TASK NAME link PROGRAM mycode
  DIR "source/tool1" SUFFIX '.c' SUFFIX '.f90'
  FILENAME "source/tool2/f1.f90"
  BUILD "obj" LIBRARY lib1 LIBRARY lib2
END
```

Again, such task looks like a compilation task with two new elements :

- the keyword PROGRAM followed by the program name,
- the keyword LIBRARY followed by the name of the static library to link with,
- the keyword DYNLIB followed by the name of the dynamic library to link with.

Local object files are also linked with the mentioned libraries to build up the executable program.

Executing operating system dependent instructions

```
STRUCTURE TASK
  #begin INST
  mkdir bin/lnux
  cp source/tool.f bin/linux/save.f
  #end
END
```

Such a task may be associated to a target and to dependencies. It is even possible to selected instruction blocks dependent on the operating system. This is one of the subjects of the article "How to".

Task dependencies

A simple project may be associated to a single task : building up the executable program from source files. But a complex project often involves several tasks, like building intermediate libraries and creating several executable program...

A configuration file may contain as many tasks as necessary and a task may be shared into several sub-tasks.

The tasks at the same level may have dependencies together. The developer is responsible of them.

Usually, a correct initial build does means that the configuration file has no mistake, even if some needed dependencies have been omitted.

Example :

```
STRUCTURE TASK NAME lib1
  DIR src1 SUFFIX '.f90' LIBNAME lib1
END
STRUCTURE TASK NAME lib2
  DIR src2 SUFFIX '.f90' LIBNAME lib2
END
```

At the first build, the library lib1 will be constructed first, just because it is mentioned first in the configuration file. But let us suppose that source files of mylib2 depend on modules generated when building up lib1.

The "normal" configuration file should be :

```
STRUCTURE TASK NAME lib1
  DIR src1 SUFFIX '.f90' LIBNAME lib1
END
STRUCTURE TASK NAME lib2 DEPEND lib1
  DIR src2 SUFFIX '.f90' LIBNAME lib2
END
```

Hopefully, even if the explicit dependency is not provided here, the modification of a critical source file of the task lib1 will induce automatically :

- the compilation of that modified source, file,
- the compilation of sources files of lib1 and lib2 using the generated module files.

Indeed, the builder takes into account, file by file, of all dependencies, not only those manages by the current task.

Such an implicit dependency, induced by the ordering of tasks in the configuration file, is only valid because the tasks are always computed sequentially. With a // task processing, such a technique would fail, critical files of lib2 being than possibly treated before recompiling modified files of lib1.

This is the reason why parallel processing is possible only within the current task, when compiling sources files of that task.

5 PLATFORM

Tuesday 8 June 2010
by FJ

```
*-> PLATFORM       s    ! general options associated to an OS and a set of compilers
   -> BUILD       c0|t  ! directory receiving constructed files
  *-> COMPILER          ! compiler description
      -> COMMAND  c0|t  ! compiler command
      -> FLAGS    c0|t  ! compiler flag
      -> OUT      c0|t  ! flag introducing the object file name
      -> INC      c0|t  ! flag followed by an include directory
      -> MOD      c0|t  ! flags followed by the directory receiving modules
      -> SUFFIX   C0    ! suffixes of the source files compiled with that compiler
  *-> DIRDEP      c0|t  ! external dependency directory
   -> DYN         c0    ! suffix of a dynamic library (.dll or .so depdending on the
operating system)
   -> DYNLIB             ! how to build up a dynamic library
      -> COMMAND  c0|t  ! main command
      -> FLAGS    c0|t  ! possible flags
      -> OUT      c0|t  ! flag introducing the library name
   -> DYNPRE      c0    ! prefix of a dynamic library (' ' of 'lib')
   -> EXE         c0    ! executable suffix
   -> LIB         c0    ! library suffix
   -> LIBPRE      c0    ! library prefix
   -> LIBRARY            ! archive description
      -> COMMAND  c0|t  ! main command
      -> FLAGS    c0|t  ! possible flags
      -> OUT      c0|t  ! flag introducing the library name
```

```
   -> NAME        c0|t  ! platform name
   -> MOD         c0    ! module suffix
   -> OBJ         c0    ! object suffix
   -> PROGRAM           ! linker
      -> COMMAND  c0|t  ! main command
      -> FLAGS    c0|t  ! various flags
      -> OUT      c0|t  ! the flags introducing the program file name
```

The PLATFORM rubric enables to define parameters independent to the project himself but essential for the build phase :

- operating system dependent parameters
- compiler parameters

A platform has a name possibly mention in the builder launching command. So it is authorized to define as many platforms as necessary in a configuration file. By default, the first platform is used.

Example :

```
STRUCTURE PLATFORM
  NAME "unix-intel"
  BUILD "bin/unix-intel"
  OBJ '.o' MOD '.mod'
  LIB '.a' EXE '' LIBPRE '' DYN '.so' DYNPRE ' '
  STRUCTURE COMPILER            ! compiler description
    SUFFIX '.f90' SUFFIX '.f'   ! source file suffixes
    COMMAND "ifort"             ! command
    FLAGS   "-O3 -openmp"       ! compilation flags
    NOLINK  "-c"                ! the special flags suppressing the link phase
    OUT     "-o"                ! the flag to precise the name of the main output file
    INC     "-I"                ! the flag to precise directories for includes and modules
    MOD     "-module"           ! the flag to precise the directory receiving modules
  END
  STRUCTURE COMPILER            ! compiler for C files
    SUFFIX '.c'   SUFFIX '.cc'  ! source file suffixes
    COMMAND "icc"               ! command
    FLAGS   "-O3"               ! compilation flags
    NOLINK  "-c"                ! the special flag preventing the link phase
    OUT     "-o"                ! the flag to precise the name of the main output file
  END
  STRUCTURE LIBRARY             ! librarian tools
    COMMAND "ar"
    FLAGS    "r"
    OUT      ""
  END
  STRUCTURE DYNLIB              ! for a dynamic library
    COMMAND "ifort"
    FLAGS    "-shared"
    OUT      "-o"
  END
  STRUCTURE PROGRAM             ! the linker
    COMMAND "ifort"             ! command to use.
    OUT     "-o"                ! the flag to precise the program file
    FLAGS   "-openmp"           ! optimizations flags
  END
END
```

6 EXCEPT

Wednesday 9 June 2010
by FJ

```
*-> EXCEPT
  *-> FILENAME    c0|t  ! particular filenames
   -> FLAGS       c0|t  ! compilation flags
   -> PLATFORM    c0|t  ! platform name
```

It is sometimes necessary to manage specific compilation options for one or several files on a given platform.

Example :

```
STRUCTURE EXCEPT
  PLATFORM "linux-intel"
  FILENAME "src/odessa_graphics.f90"
  FLAGS "-O0"  ! because of a compiler trouble with the options -O3 -openmp
END
```

7 How to

Wednesday 9 June 2010
by FJ

The ODESSA data reader authorizes many operations which may be useful for an experienced user

Introducing parameters in the configuration file

The ODESSA data reader is connected to another ODESSA tool called *Analyzer*. This analyzer is itself a true programming language with many operators.

The coupling reader/analyzer is managed by small instructions between parentheses. When the reader meets a string between parentheses, it calls the analyzer to execute this short instruction. Such an instruction may have a result or not.

Example :

```
(debug='-g')

STRUCTURE COMPILER
  NAME ifort
  FLAGS (debug)
  ...
END
STRUCTURE COMPILER
  NAME icc
  FLAGS (debug)
  ...
END
```

The first analyzer analyzer instruction is (debug='-g'). This instruction has no direct result for the reader which sees nothing but the analyzer creates a variable named debug and having the value '-g'.

The second analyzer instruction is (debug). The analyzer just return the value of the

variable *debug*, i.e. the string '**-g**'.

The analyzer is a rather complicated language but, in the framework of builder configuration files, only few operators must be known :

=	creation of a variable. For instance : (a=3)
:	extracting something from a composed object
//	merging two strings
==	comparison operator as in FORTRAN or C
<>	comparison operator like /= in FORTRAN and != in C
AND	logical operator equivalent to .AND. in FORTRAN and && in C
OR	logical operator equivalent to .OR. in FORTRAN
DELIMITE	'/' on UNIX-like systems and '\' on Windows
GETENV	to get an environment variable

Of course, arithmetic operators + - / * are also available as well as mathematical operators like ** SIN COS TAN EXP LOG... but their interest is very limited here.

Defining a parameter from command line

Rather than initializing the variable **debug** in the configuration file, it is possible to do it via the command line :

```
builder -p debug '-g' ...
```

The configuration file must be modified as follows :

```
(IF(1-ASSIGNED('debug')) debug=' ')
```

It means that the parameter **debug** is initialized to ' ' by the configuration file only if its is not already defined via the command line.

In the predefined platforms, several parameters may be defined via the builder command :

- FC : Fortran compiler
- CC : C compiler
- FFLAGS :Fortran compiling flags
- CFLAGS : C compiling flags
- LDFLAGS : link flags

How to re-use platform blocks prepared for another project

The ODESSA data reader offers several possibilities :

- it accept the keyword CALL followed by a file name. This command includes that file into the configuration file.
- it is possible to prepare parametrized platform blocks.

For instance, the ODESSA builder is delivered with predefined platforms : linux-intel, linux-gcc, linux-g95, linux-nag, windows-intel, windows-gcc, windows-g95, windows-nag. These platforms have been parametrized :

- the platform name,
- predefined flags for the compilers and the linker : fflags, cflags, ldflags

For instance, if you want to load the platforms linux-intel and linux-gcc under two variants, debug and optimized versions, then the configuration file could be :

```
STRUCTURE TASK ... END
...
(debug = '-g')
CALL linux-intel.cfg

(debug=' ')
CALL linux-intel.cfg

(debug='-g')
CALL linux-gcc.cfg

(debug=' ')
CALL linux-gcc.cfg
```

Let us notice that a same file may be included several times.

It is even possible to group together all the platforms into a single file and to call that file in your configuration file : look into the file odessa/dat/platforms.cfg.

The configuration file becomes :

```
CALL platforms.cfg
STRUCTURE TASK ... END
..
```

The ODESSA data reader always tries to open the include files locally. In case of failure, it tries to open them from the ODESSA directory.

How to test operating system parameters when describing tasks

All has been done for avoiding specific platform or OS features in the description of tasks. Unfortunately, all is not always enough !

So additional possibilities have been added which makes possible to select specific features depending on the chosen platform.

The advised way

It is possible to use COND data blocks which contain a special datum named IF and followed by a text (string between quotation marks). This text represents an instruction which will be executed **after** the complete reading of the configuration file.

This short instruction has to return a logical result (true or false). If the condition is true, then all the data within the the condition block become data the parent block, else they are simply ignored.

To help the developer, the current platform is a datum which can be tested. Example :

```
STRUCTURE TASK NAME "screen_driver"
  STRUCTURE COND
    IF "platform:'NAME': 1 5 == 'linux'"
    DIR "screen/x11"
    ...
  END
  STRUCTURE COND
    IF "platform:'NAME': 1 5 == 'windo'"
    DIR "screen\japi"
    ...
  END
  ...
```

Two conditional blocks are described here. The first one selects data for a Linux platform whereas the second one selects data for the Windows platform.

Such conditional block is not very flexible. For instance, it does not have a "else" counter part.

About the conditional instructions :

- the first one is `platform:'NAME': 1 5 == 'linux'`. The operator : has a high precedence level and is executed before ==. The first : extracts the platform name whereas the second : extracts from that name the substring composed of the 5 first characters. == compares that substring with the string 'linux' and return true or false.
- the second condition is very similar to the first one.

Let us notice that these instructions are executed by the ODESSA analyzer. They are not between parentheses to avoid their execution during the reading phase.

Testing the directory path delimiter to know the type of operating system

This technique may be applied when reading the configuration file. The ODESSA data reader has special keywords #ifthen #elseif #else and #endif to selected data under condition.

```
STRUCTURE TASK NAME driver
  #ifthen(DELIMITE == '/')
    DIR "screen/x11"
    ...
  #else
    DIR "screen/japi"
    ...
  #endif
  ...
END
```

Notice that the result is very similar to the one obtained by COND blocks. The main difference is the timing, COND block being analyzed after the reading and also after a check phase verifying that the configuration file has no mistake.

This kind of technique is used to install ODESSA itself (see the file odessa/dat/odessa.cfg)

Getting parameters of the current platform

It is even possible to go further in the way of the previous paragraph.

Let us assume that all the platforms are described **before** the tasks in the configuration file. The builder argument is always passed to the data reader under the name **argument**. It is therefore possible to get the associated platform rubric and to use the platform data as

parameters when reading the tasks.

```
CALL platforms.all

! getting the platform rubric

#ifthen(argument == "")
  (platform=GLOBAL:'PLATFORM' 1)
#else
  (platform=GLOBAL:'PLATFORM' argument)
#endif
```

GLOBAL is the data base containing the data of the configuration file which have been read up to now. The operator **:** extracts a piece of information from a data base.

After the previous sequence, it is possible to extract from the variable **platform** all the data it contains. For instance, (platform:'OBJ') is the string containing the suffix of an object file, (platform:'BUILD') is the building directory and (platform:'NAME') is its name ...