The Real Programmer Tool
a set of procedures for Fortran programmers

software and documentation
prepared by L.Chiappetti - IFCTR
version 1.6
Jun 1993

Real Programmers do it in Fortran
The determined Real Programmer can write Fortran programs in any language

Table of content

software and document history

Aug 90	preliminary version (1.0) of the software (not all functions) and draft version of the documentation
Feb 91	final version of the software (all functions 1.0), including correction of a few bugs (1.1). Software not publicly installed.
Apr 91	Final version (1.0) of documentation, and software publicly installed.
Dec 91	Documentation converted to Word (1.1), fixed a few bugs and support for DECstation in software (1.2). Single-barred changed paragraphs.
Apr 92	Unified Sun and DECstation version of compile (1.3), rearranged handling of include files. Double-barred changed paragraphs.
May 92	VMS version of ptlink (1.2) changed to use a temporary `LNK$LIBRARY` to support multiple search of unresolved references. Thick barred changed paragraph.
Jun 92	VMS version of compile handles also *.f files for Unix compatibility; Unix version of compile handles include files differently (this way entire VMS source directories can be NFS mounted on Unix machines). Thick barred changed paragraph.
Sep 92	(VMS) makeknown (1.1) also support `$PATH`
Jun 93	compile 1.5 now supports nested include files

1. Introduction

Have you ever needed a way of compiling and linking your programs with simple, system independent commands, without passing all the times long string of options ? Have you ever wished to be able to edit, compile, link and run programs, when the source, object, library and executable files all (may) reside in different directories, and may be the data reside in another directory again. Have you ever got fed up of typing long pathnames, or changing continuosly directory, or struggling with complicated directory trees where you can never find the library subroutine you need ? Or just do you want a tool to do all that, and orderly manage your programs ?

The "Real Programmer Tool" is for you

The basic logics under this set of procedures is the following:

Have the main program sources *somewhere*.

Have the sources for library routines in one subdirectory with the name of the library under *somewhere else*.

All sources make reference to include files by relative names (ensuring source-level code portability). These include files are stored *somewhere else again*.

Object files are by definition scratch.

Library subroutines are kept in object libraries under *somewhere else again*.

A set of utilities will find all this, and put the executables in the *right place* and make them available (this essentially means in VMS you no longer need to type `RUN` but can use the program name as a full fledged command with parameters).

You will be able to do all that (including editing the different source files), *irrespective* of what your current directory is (may be that is your data directory, for instance).

A small set of simple procedures (available for VMS and Unix under a very similar interface) is presented here. The purpose of such procedures is to aid the Fortran programmer to manage his files (source, relocatable, libraries and executables) in an orderly way, without however resorting to system-specific or mysterious tools (e.g. make). It is of course not the only way of doing it, but is a way I am used to.

The procedures have characteristics similar, for those who know it, to the tools developed for the Milan Exosat system on the IBM (namely `COMPILA`, `COMLINK` and `COMPLIB`), with however substantial additions to handle the new possibilities offered by hierarchical directory systems (like VMS and Unix). Conversations with B.Garilli are gratefully acknowledged.

2. Essential capabilities

All the commands described below can be made available by executing once a simple procedure **progtool**[1], which defines all the other commands. If an user wishes to have them always available it is sufficient to call it within the login file (`LOGIN.COM, .cshrc, or .mycshrc`).

[1]: commands are presented in lower case for generality, except whenever upper case is explicitly required. VMS users can use any case (VMS is case-insensitive).

The essential commands to handle compilation and linking have been prepared according to the following guidelines (similar to the above mentioned IBM `EXEC`s) :

A single command **compile** should be used for compilation (either of main programs or of library routines). A set of standard, useful compilation options shall be generated automatically by this command. The user will not have to deal with long and awkward compiler command strings.

Additional options of common usage under some contexts (to be determined by the user) can be stored once for later use, by the command **options**. In particular the optimization level (for optimizing compilers) can be stored with an unique syntax by command **optimize**.

For what concerns terminal display options, as default operation, the compilation command will compile *silently*. If this is not the case, two sets of standard options are available to allow:

display of compiler messages on the terminal (option **term**)

display and browsing of messages as well as of the full program listing (option **list**)

A command **comlink** is used to compile and link in one go. The object file is then deleted. Although compilation-with-linking is the default action under Unix compilers, it is felt more "didactic" that the user be aware of the difference between the two steps, in the old JCL spirit of the "CLG" (Compile, Link and Go) steps. (JCL, the IBM OS Job Control Language, is considered here like the "Latin" of computers ... not that Unix is such a great leap forward for what hydiosincracies about syntax is concerned !). This way the same compilation command can be used for main programs and for library subroutines. In fact **comlink** calls in turn **compile**, the linking command described below, and an internal cleanup procedure.

A standalone command for linking is available. This is currently called **ptlink** to avoid naming clashes with the system "link" command. Both ptlink and comlink accept as argument a *loader file*, which contains the commands for the linker.

In the current version this loader file just contains a list of libraries. Support to other

options will be arranged in the future.

A command **complib** is used to compile library routines and insert/replace them in an object library. The same command is used to recompile entire libraries. The intermediate individual object files are deleted.

As underlying philosophy it shall be noted that it is assumed here that subroutines can be found only in two places : within the same source file as a main program, if they are called only by that particular program; or in a library. There will not be independent subroutine object files !!

The case of library subroutines which have to be relocated together (e.g. BLOCK DATA), and the case of INCLUDE files may require special handling to fulfill the requirement of source-code-level portability. This is described below.

The additional commands added to take advantage of the capabilities of the hierarchical directory systems (but to avoid messing up and getting lost in the directory tree) are designed to define (and store in a "global" area) standard "places" where the other commands will look. In particular one can edit source, include and library source files (irrespective of the "current" directoy where one is) with the **edfor**, **edinc** and **edlib** commands.

In designing the additional commands, I have kept present the need for flexibility. The user is not forced to a rigid scheme, but can choose his preferred scheme (e.g. a set of "parallel" directories, or a tree, or even keep everything in a single directory), or even a different scheme according to each project he is working on.

These commands are of the form *command pathname*, where the command is one of the mnemonics listed below, and *pathname* is either a valid pathname under VMS or Unix, or (for the VMS case) a pathname in the (Unix-like) form supported by the UNIQ interface (see "The UNIQ interface. An unique command interface for VMS and Unix users", L.Chiappetti, December 1990; and also section 3.2 below). In all cases pathnames are translated in *absolute* pathnames.

The standard places defined are the following :

sourcedir the directory where the Fortran source files of main programs are to be found

libsourcedir a directory which should contain one subdirectory for each library. Such library subdirectory will contain the individual Fortran source code of each library routine.

incdir a directory where source files required by INCLUDE statements are to be found

relocdir he directory where the relocatable object files (product of compilation) will be put. This could be a temporary area, as such files are deleted after linking

or insertion in a library.

libdir the directory where complete relocatable libraries are to be kept.

targetdir the directory where executables (product of linking) will be stored.

If wished the user can define a standard setup of directories in a command file, and execute it in his login procedure. Alternatively he can keep project-related setups in different command files, and execute the relevant one when needed. Unix users shall note that such files have to be sourced.

Handling of INCLUDE files is currently supported with a precise limitation, due to the requirement of portability at source code level. The only form of INCLUDE statement which looks portable is the one :

INCLUDE '*filename.type*'

without any path specification. The *filename* shall be lowercase, the type shall be .inc and the file shall reside in the *incdir* (or it shall reside in sourcedir if another type is used).

Incidentally, due to the highly system-dependent way compilers handle INCLUDE statements, this means that Fortran programs with such INCLUDE statements (and the include files located in a directory other than the *sourcedir*) could generally be compiled only using the tools described here.

Concerning library names see the "note on library naming" below in 3.6.2.

An appendix gives some possible directory arrangements as examples.

3. User's guide to commands

Real Programmers do it in Fortran

The following conventions are used:

A word in **boldface** shall be typed exactly as it is (command name or option).

A word in *italic* shall be replaced by a valid parameter. A parameter in parenthesis is optional (the parentheses shall NOT be typed).

3.1 making the commands available

Just issue the command **progtool** once (e.g. in your login file).

3.2 defining standard directories

Issue, in any sequence, one or more of the following commands (a good place to put them is a `.COM` file, or Unix script to be sourced, in the top directory of the software project : I use a file called `CONFIGURE.COM` or just `configure`):

sourcedir (*argument*)
libsourcedir (*argument*)
incdir (*argument*)
relocdir (*argument*)
libdir (*argument*)
targetdir (*argument*)

where the *argument* can be one of the following :

none — in this case the current value is displayed

clear — in this case the current value is reset to "undefined" (as it is if the relevant command has never been entered)

dir — a valid pathname of one of the following forms :

Unix:
- an absolute pathname */dir/dir*
- a relative pathname *dir/dir*
- a relative pathname *./dir/dir*

VMS:
- a VMS pathname *dev:[dir.dir]* in any valid VMS form (that is any of the elements can be missing, and any abbreviation like [-], [--] can be used).
- a VMS logical name pointing to a valid pathname

VMS Uniq:
- a *name* (translated into *[.name]*, a subdirectory of the current directory)

a \n*ame* (translated into *[name]*, an absolute pathname on the current disk)
a *device:name* or *device:\name* (translated into *device:[name],* an absolute pathname on an alternate disk
In all the above cases, each name can be a path of further subdirectories or the form *dir\dir*, i.e. Unix-like but backslash-separated (the slash is a reserved character in VMS).

For more details about the Uniq directory naming, see section 1.1 of the "Uniq interface Users' Guide" (quoted above).

In all cases the pathname should be translated into a full pathname. Currently relative names of the form *../name* do not work under Unix : it is recommended to give absolute path names (*/name*), or paths relative to the home directory (*~/name*). In all cases it is verified that such directory exists (if it does not, the current definition is not changed).

The current definition is kept in global symbols (VMS) or environment variables (Unix) with upper case names prefixed with **PT_** (this way the command **sourcedir** generates **PT_SOURCEDIR** etc.).

If you want, for project purposes (or just to define a "standard configuration") to include one or more of such commands in a procedure, you may do it freely under VMS .COM files, but in the case of Unix you shall source such procedure, not call it as a shell script, if you want that the settings are preserved in the environment of the shell from which you are calling the procedure.

3.3 defining options for the compiler

If you want to define options other than the default ones described in 3.4 below (or to override them), you may take advantage of the following procedures. If you execute them once the compile command will remember them until they are explicitly reset.

3.3.1 defining the optimization level

To choose the optimization level for all subsequent compilation, issue the command:

optimize (*argument*)

where *argument* can be one of the following :

none	he current setting is displayed
clear	the current setting is reset to "undefined"
0	or no optimization (Unix and VMS)
OFF	for no optimization (VMS only)
n	with n equal to **1**, **2** or **3** (under Unix this sets the corresponding

optimization level, under VMS any value enables optimization)

ON to enable optimization (VMS only)

any any other value enables optimization under VMS, and disables it under Unix (i.e. reverts to the default behaviour of the compiler).

3.3.2 defining other options

If you want to define additional, system dependent, options in a way that you do it once, and they are remembered until reset, use the following command:

options (*argument*)

where *argument* can be one of the following :

none the current setting is displayed

clear the current setting is reset to "undefined"

string a valid string of options in the syntax required by your compiler (with the exceptions noted below).

VMS: you can prefix your options with the slash (/) required by the compiler, or not do it (provided you separate different options with blanks). **options** will generate the correct syntax. Note this means that if any option requires arguments the latter shall NOT be separated from the option by any blank.

You cannot specify more than 8 options (DCL limit)

Unix you can prefix your options with the minus (-) required by the compiler, or not do it (but you shall always separate different options with blanks). **options** will generate the correct syntax. Note this means that if any option requires arguments the latter shall NOT be separated from the option by any blank.

Note that in no case you are allowed to add options to the current list: they are overwritten with the new ones. You have to retype explicitly any options you want to preserve.

3.3.3 common features

Optimization level and other options are stored in global symbols (VMS) or environment variables (Unix), called respectively **PT_OPTIMIZE** and **PT_OPTIONS**. See the note at the end of 3.2 for the usage of the commands above within Unix scripts (they have to be sourced).

Please note that **optimize** does not necessarily store the options in the format used by the compiler.

3.4 compiling a program

To compile a Fortran source file, you use the following command (however in the majority of cases you'll want to compile and link together, see **comlink** in 3.6 below) :

compile *program* (*option*)

where *program* is the name of a Fortran file (without the .FOR or .f filetype specification), which shall exist in the appropriate place, and *option* may be one of the following :

none — or default behaviour. Compilation is silent, all messages are suppressed (like IBM VS Fortran `NOTERM`), except a single warning by **compile** in case of errors. Typically used when recompiling a known program.

term — enables compiler messages to be displayed on the terminal. Typically used when compiling a new or modified program, and you are sufficiently sure that no severe errors will occur.

list — if you want to inspect a listing file. Used for debugging. See below for system dependent details.

The command expects a file *program***.FOR** (VMS) or *program***.f** (VMS and Unix) to exist in one of the following two places : the current *sourcedir* if defined, or the current directory otherwise.

The command creates an option string based on the user setup (commands **optimize** and **options**) and on the following internal defaults :

VMS: `SHOW=(INCLUDE,NOMAP,NOSINGLE)`
Unix: `-c` (to prevent linking, do not reset)

The command creates an object file *program***.OBJ** (VMS) or *program***.o** (Unix) in one of the following two places : the current *relocdir* if defined, or the current directory otherwise.

If listing is required any temporary file is created in the same place as the object files, and deleted on exit.

For VMS, the *program***.LIS** file is "browsed" entering into EDIT/READONLY (use normal editing commands to move around). The editor is always EDT, irrespective of any other private setting of the symbol EDIT.

For Unix on Sun, the Sun Fortran compiler has no listing option. This is emulated as

follows (under SunView and OpenWindows only) : *two* windows are opened, one containing the original source file, and one containing the redirected output messages of the compiler. Both windows use vi in readonly mode. This choice (instead of the nicer textedit) is due to the fact that vi allows to display line numbers (useful to locate errors in the source file). To continue one should quit the source file (yellow) window (typing the vi command **:q**). The other window is automatically killed. As additional aid, note that the error message window is either green (no errors) or red (errors encountered).

For Unix on DEC Ultrix, the DEC Fortran compiler has a listing option (-V), and may produce *program***.l**. Browsing is emulated as follows (under DECWindows only) : *one* windows is opened, containing the listing (inclusive of error messages). The window uses the notepad editor. To continue one should quit the notepad window (e.g. keying **Alt-Q**). Please be sure to quit the editor, otherwise you will be left with a *program*.**l.BAK** liying around. As additional aid, note that the listing window is either green (no errors) or red (errors encountered).

3.4.1 handling include files

The code of main programs and library routines shall make reference to include files (which should exist in *incdir*) by statements like :

INCLUDE 'filename.type'

only. No path name shall be given if you want to preserve portability of the code (as path names are system dependent). Incidentally (and unfortunately) this means I could not use .TLB text libraries to handle include files under VMS. (If you are not concerned about portability this is not a problem for you).

The type of the include file shall be specified explicitly as there is no agreed system-independent standard or default. If you want portability, you cannot rely on the type being .FOR or .f. Note that the existence of the include file is checked only at compile time (therefore if it does not exist, this generates a compilation error).

For elegance, compatibility with the XAS convention and ease of use with the trick developed to cope with the limitations of the DEC Ultrix compiler (see 4.4 below) , it is compulsory to use type .inc for include files (unless they reside in *sourcedir*).

The **compile** command does a few (transparent) changes of directory (described in 4.4 below) to cope with the above policy for include files. This essentially means that you must use the progtool utilities to compile programs written with include files (this matters if you are giving your code to external sites), unless the include files reside in the same directory as the source code (that is *incdir* is equal to *sourcedir*).

3.5 linking a program

To link an object file produced by **compile**, you use the following command (however in the majority of cases you'll want to compile and link together, see **comlink** in

3.6 below) :

ptlink *program* (*loaderfile option*)

where *program* is the name of an object file (without the .OBJ or .o filetype specification), which shall exist in the appropriate place, *loaderfile* is the optional name of a file of instructions for the linker (necessary only to specify libraries or other special requirements) and *option* may be one of the following : none at all, **term** or **list**.

option applies to VMS systems only. If any (non null) option is specified, linker error messages are displayed on the terminal. The Sun Unix linker does not display messages anyhow (except if a map option was passed down by the Fortran compiler). In all cases the **ptlink** command will display an own message telling whether errors have occurred.

Since this command will seldom be used alone, for all other parameters refer to the **comlink** command below.

3.6 compiling and linking a program

To compile and link a Fortran program in one go you use the following command :

comlink *program* (*loaderfile option*)

where *program* is the name of a Fortran source file (without the .FOR or .f filetype specification), which shall exist in the appropriate place, *loaderfile* is the optional name of a file of instructions for the linker (necessary only to specify libraries or other special requirements) and *option* may be one of the following : none at all, **term** or **list**.

The *loaderfile* is discussed below. To call **comlink** with compiler options, but without loader file use :

comlink *program* **""** *option*

This command executes in turn : **compile** for compilation (see 3.4 for details); **ptlink** for linking (see 3.5) and its own internal cleanup procedure.

For the meaning of the *option*s, which are passed to the procedures called and are essentially used by **compile** only see 3.4 above. Any non-null option will also cause linker messages to be displayed on the terminal on VMS systems (they are otherwise disabled; the Sun Unix linker does not provide any message, except if a map option is inherited from the compiler).

compile expects a file *program*.**FOR** (VMS) or *program*.**f** (VMS and Unix) to exist in one of the following two places : the current *sourcedir* if defined, or the current directory otherwise.

See 3.4 above for a discussion of compilation options.

compile creates (and **ptlink** expects) an object file *program***.OBJ** (VMS) or *program***.o** (Unix) in one of the following two places : the current *relocdir* if defined, or the current directory otherwise. The object file is always deleted on exit by **comlink** (but not by **ptlink**).

If listing is required any temporary file is created by **compile** in the same place as the object files, and deleted on exit.

ptlink creates an executable file *program***.EXE** (VMS) or *program* (Unix) in one of the following two places : the current *targetdir* if defined, or the current directory otherwise. In the case of VMS systems, **comlink** purges all older versions of the executable, in the case of succesful linking only.

In the case of VMS only, **comlink** calls **makeknown** at the end. This auxiliary utility may also be called standalone (not normally done) with the following syntax :

makeknown *program* (*path*)

Essentially this utility defines a symbol program pointing to the image *path:program***.EXE**, which is therefore installed as a VMS foreign task. Note that this occurs only if the symbol does not already exist with a different value !

If a logical **$PATH** is defined as a list of directories, and this includes the path argument, **makeknown** will generate a symbol pointing to **$PATH**:*program***.EXE**

If **makeknown** is succesful, the program can be invoked without `RUN`, just by typing the name (and passing parameters is enabled).

Note that *path*, if not specified, defaults to current *targetdir*. Also note that *path*, if given, shall be in full VMS form. Uniq or partial forms are not recognized (normally they have already been translated by **comlink**). If you really have to use it, express path always in the full form *device***:[***dir.dir***]**.

In the case of Unix, *targetdir* shall be in the search path care of the user in order to invoke the command (a rehash might be necessary anyhow).

3.6.1 Usage of loader files and libraries

If a loaderfile is specified as second argument to **ptlink** or **comlink**, this means a file loaderfile**.LOADER** (VMS) or loaderfile**.loader** (Unix) shall exist in either the current sourcedir if defined or the current directory otherwise.

In the current implementation only the first (only) line of such file is used. Such file will

contain a list of relocatable library names, in the correct order required by the linker.

These names shall be in the format described in 3.6.2 below, and are used to construct the run string for the linker. The libraries shall by default exist in one of the current two places : the current *libdir* if defined, or the current directory otherwise. If however a pathname is prefixed to the library name, this is used to construct the run string.

The result of this may be system dependent and has not been tested.

ptlink does not check for the existence of libraries. If they do not exist, a linking error will be signalled by the system linker.

In the case of VMS the linker command string is constructed in the following way:

for each library without explicit pathname, a pathname is constructed using either *libdir* or the current SET DEF.

such final library names are used as input to the linker, with the `/LIBRARY` qualifier.

the entire runstring is appended with the following qualifier: `/EXECUTABLE=`*path*`:`*program*, where *path* is either *targetdir* or the current SET DEF.

In the case of Unix the linker **ld** is not called directly, but through the **f77** Fortran compiler command (which receiving an .o file as input understands it has just to call the linker). It is this command which takes care of constructing the appropriate options (and standard Fortran library definitions) for Fortran programs. Additional options (passed thru by f77 to ld) are constructed as follows :

the current *libdir* (or the working directory) is used to set up an option **-L***libdir*.

for each library *lib* without explicit pathname, an option **-l***lib* is added to the option string.

each library name with an explicit pathname is separated into a dirname and basename, used to set up separate **-L** and **-l** options

finally a **-o** options pointing to the executable file is set up

3.6.2 Note on library naming

There is a difference in the way the VMS and Unix standard tools understand library names. Therefore it is necessary to explain how this is interpreted here. A library name of *lib* is used as follows:

In VMS :

The relocatable library shall be a *lib***.OLB** file, located in the current *libdir* directory.

The sources of the library subroutines shall be located in a subdirectory *lib* of the current *libsourcedir* directory.

Example: if *libsourcedir*=`DUA0:[LUCIO.FORTRAN]`, *libdir*=`DUA0:[LUCIO.LIB]`, library `TESTLIB`, then the source for the hypothetic routine `SUB1` is in the file `DUA0:[LUCIO.FORTRAN.TESTLIB]SUB1.FOR`, and the entire relocatable library in `DUA0:[LUCIO.LIB]TESTLIB.OLB`.

In Unix:

The relocatable library shall be a **lib***lib***.a** file, located in the current *libdir* directory.

The sources of the library subroutines shall be located in a subdirectory *lib* of the current *libsourcedir* directory.

Example: if *libsourcedir*=`/home/lucio/fortran`, *libdir*=`/home/lucio/lib`, library `test` then the source for the hypothetic routine `sub1` is in the file `/home/lucio/fortran/test/sub1.f` and the entire relocatable library in `/home/lucio/lib/libtest.a`.

Note that if I had called the library `testlib`, the relocatable library would be `/home/lucio/lib/libtestlib.a`

3.7 compiling a library or library member

To compile a library subroutine and insert it into the appropriate library use the following command :

complib *library member* (*option*)

while to compile the entire library (see 3.7.1 for details) use instead:

complib *library* **all** (*option*) (Unix,VMS) or
complib *library* **ALL** (*option*) (VMS) or
complib *library* ***** (*option*) (VMS)

where *library* is the name of a relocatable library (as described in 3.6.2 above; without any **.OLB** (VMS) or **.a** (Unix) filetype specification and without the Unix **lib** prefix), which shall exist in the appropriate place, *member* is the name of the subroutine Fortran source file, which shall exist in the appropriate place, and *option* may be one of the following: none at all, **term** or **list**.

Options are passed on to the **compile** procedure and are described in 3.4 above.

complib uses **compile** for compilation (however it resets temporarily the sourcedir specification, to allow library routine sources to reside in a different place than main program sources). It also handles include files identically as **compile**.

complib (in single member mode) expects a file *member***.FOR** (VMS) or *member***.f** (VMS or Unix) to exist in a subdirectory *library* of either : the current *libsourcedir* if defined, or the current directory otherwise (this implies the subdirectory itself exists).

compile called by **complib** creates an object file *member***.OBJ** (VMS) or *member***.o** (Unix) in one of the following two places : the current *relocdir* if defined, or the current directory otherwise. The object file is always deleted on exit by **complib**.

If listing is required any temporary file is created by **compile** in the same place as the object files, and deleted on exit.

complib expects a file *library***.OLB** (VMS) or **lib***library***.a** (Unix) in one of the following two places : the current *libdir* if defined, or the current directory otherwise. If this file does not exist, it is created explicitly (VMS) or automatically by `ar` (Unix).

complib, in the case of succesful compilation, will insert the member into the library (replacing any previously existing member of the same name). In Unix `ranlib` is run at the end.

3.7.1 compiling an entire library

To compile an entire library specify a member of **all** or *****, and note what follows.

The * option, although designed for analogy with Unix syntax, is currently disabled under Unix, due to unresolved conflicts with the variable and filename substitution mechanisms.

Under VMS at least one **.FOR** file shall exist in a subdirectory *library* of either : the current *libsourcedi*r if defined, or the current directory otherwise. Under Unix this test is disabled due to the above mentioned conflicts and only the existence of the subdirectory is checked.

complib then proceeds in an identical manner as in the single member case, except that :

if the library exists already, the old version is first of all renamed (the type is changed to **.OLDOLB** (VMS) or **.old** (Unix))

each member is compiled in turn, any compilation errors are signalled, but the procedure is not interrupted. The wrong member is not inserted in the library.

however an error inserting a member (Unix only) causes termination

`ranlib` (Unix) is run only once after all compilations and insertions are done

if no errors occurred, the old version of the library (if any) is deleted

if errors occurred, and **complib** is running in interactive mode, the user is presented to options : keep the new version and delete the old one; or restore the old one (delete the new one, and rename back the old one)

if errors occurred, and **complib** was run non-interactively, the old version is restored by default.

The definition of interactive mode is rather tricky. In VMS it means that `COMPLIB.COM` has a depth of 1 (that is, is being called by DCL and not by other `.COM` file). In Unix it should mean the same thing, that is `complib` is called by `csh`, and not by another script, but the implementation is not guaranteed, as there is no easy way of telling that. Note that with the present definition if `complib` is run in background, it will appear to be "interactive". The case of batch mode has not been tested.

3.8 Editing source, include or library source files

Three little utilities are provided to edit source files, include files, and library source files in their respective *sourcedir*, *incdir* or *libsourcedir* directory, starting from anywhere. Their syntax is :

edfor *file*
edinc *file.typ*
edlib *library memberfile*

The parameters are self-explanatory. A file type of **.FOR** or **.f** is appended for source and library source files, and therefore the type shall not be specified. On the contrary, it is necessary to specify a type for include files (at least until a convention on this will be reached, then it could be enforced here).

The editor used is whatever the user gets when he calls "edit" (this assumes he has defined a symbol of this name to set up his private choice, or that a reasonable choice is set up for him by the system).

4. Programmer's notes

The determined Real Programmer can write Fortran programs in any language

These procedures will be moved in `DUA0:[LOCAL.PROGTOOL]` or `/usr/local` from their current development directories in `DUA0:[LUCIO.BIN.PROGTOOL]` and `/home/lucio/bin`, as soon as they are officially released (contextually with the issue of the present document). External sites wishing to install the software shall check the dependency on such disk names in the code.

The Unix version has one external dependency, on the shell script `windowsystem`, used to determine which window system (Sunview, OpenWindows, DECWindows or none) one is using.. The VMS version has the following dependencies on external modules : `PROGTOOL.COM` uses the `KNOW.COM` utility (part of XANADU) to set up symbols. `SETXXXDIR` uses `VMSTRANSLATE.COM` and `VMSVALIDDIR.COM` (part of the Uniq interface). Most modules use the symbol `VMSDELETE` to delete files (this is just an alias of the usual VMS `DELETE`, which is set up by the Uniq interface, as the latter redefines the name `DELETE`).

4.1 progtool

VMS version of progtool is `PROGTOOL.COM`. Shall be defined as known in `SYLOGIN`. It uses the `XANADU KNOW` command to make known all other `.COM` files (all commands but the "set directory" family). Note the syntax used to call a symbol when defining a symbol.

Unix version shall be aliased to "`source progtool`" in `.cshrc`, otherwise the current command has to be issued twice (first time as procedure, second time as alias), or sourced explicitly. Most commands are defined as aliases. The others shall be available if `/usr/local` is in the path.

4.2 sourcedir, libsourcedir etc.

VMS versions are global symbols pointing to an hidden `.COM` file called `SETXXXDIR`, which receives an additional argument of `SOURCE` (for source), `LIBSOURCE` (for libsource), etc.

It calls on its turn `VMSTRANSLATE.COM` when translation from the UNIQ to the VMS name form is needed, and `VMSVALIDDIR` to test existence of directories. They communicate through a (temporary) global symbol `VMST_RETURN.` These utilities are part of the Uniq interface software and are not described here.

The directory paths are stored in global symbols (`PT_SOURCEDIR`, etc.)

Unix versions are aliases set up by `progtool`. Note that the hidden `setxxxdir`

script (similar to the VMS equivalent described above) has to be sourced. As such it cannot receive arguments, and these have to be passed via (temporary) environment variables.

The reason they have to be sourced is that the directory paths are stored in environment variables (`PT_SOURCEDIR` etc.), and these are not passed back to the parent shell by a script.

The check on valid directories is easily done internally via Unix primitives.

Note the tricky way of checking whether an environment variable exists, when the name of the variable is a complex expression constructed within the procedure.

4.3 optimize and options

VMS versions are self-explanatory `.COM` files. The relevant info is stored in global symbols.

Unix versions are aliases set up by `progtool`. As described in 4.2 they have to be sourced and receive arguments thru temporary environment variables, because the relevant info itself is stored in environment variables (which are not passed back to parent shell by a script).

This causes some minor complications (use of parentheses) with the argv vector in options.

The loop is better handled (while ...) in Unix than in VMS.

Note sometimes an if then else is needed instead of a single if, because the latter is fully constructed before being tested ... in which case something like if (variable exists) then expr(variable) of course cannot be constructed.

4.4 compile

VMS version is a `.COM` file. The `ON ERROR CONTINUE` is necessary to continue execution and trap return code after an erroneous compilation. Note the use of lexicals, the tricks to check file existence with a "null" `DIR`, and the way to set up the editor in screen mode (also this editor is always `EDT`, which means any user private definition of the symbol `EDIT` is temporarily overridden and restored later). Now for Unix compatibility, if no .FOR file exists a .F file is tried before giving a "file not found" error".

Unix version is a csh script. Note the extensive use of parentheses for multi-word variables, and the usual if then else problem.

Output redirection is used to route the compiler messages to `/dev/null` (none), the terminal, or a temporary file.

The only way to force the object file into a particular directory is to `cd` there before running `f77`. This is done since version 1.3 and is compatible with the handling of include files (see below)

The Sun `f77` compiler has no "list" option, and this is emulated (see discussion in user section above). Note the use of `kill %` to terminate the window containing the compiler messages.

Handling of include files is now supported. The basic requirement is that source code shall be ported between VMS and Unix systems (and v.v.) unchanged, without the use of any preprocessor. Unfortunately the Fortran `INCLUDE` statement is not standard (essentially this derives from the fact it references system-dependent filenames, but there is more to that : on IBM systems for instance `INCLUDE` cannot reference files, but only members of a macro library, with syntax `INCLUDE` (*member*); this parallels one of the possibilities offered by VMS, that is referencing a member of a text library, with syntax `INCLUDE` '(*member*)'; unfortunately there is nothing of this sort under Unix, which provides extra flexibility allowing C-like `#include` directives, which are obviously not portable).

The only way of having a portable `INCLUDE` statement is to specify a pathless (relative) file name, with the syntax `INCLUDE` '*filename.typ*'. One cannot obviously use absolute paths, as these are system-dependent. However also the way compilers look for relative include files is system dependent.

All the above will work and be portable (even without using the progtool utilities) if the include files resides in the same directory as the program source which references it.

However in our case this is not generally wished : we would like to have include files in a common *incdir*, and be reference from main programs in *sourcedir*, as well as from library routines in *libsourcedir*. If all these directories do not coincide one must use the progtool utilities.

The VMS implementation involves doing a SET DEF to the *incdir* temporarily just before calling the compiler. The paths for all other files (source and object) can be given explicitly on the command line.

In Unix things are trickier. Some of the places the compiler looks for include files (subdirectories of where the main resides, or `/usr/include`) are not portable. Under SunOS we originally did a `cd` to the *incdir* temporarily. Unfortunately this had a side effect, that is the object file will be created there (its path cannot be specified). Therefore a corrective action was necessary to move the relocatable to the *relocdir* (a plain move with no overheads if the two directories are on the same filesystem or a physical copy if they are on different filesystem with an overhead in execution time and disk space, perhaps inconvenient for large programs). In DEC Ultrix Unix the compiler never looks in the current directory, therefore cd to the *incdir* does not work. There is no way of specifying a separate path for include files. They have to be in the *sourcedir*.

The only way which works in both systems (implemented since 1.3) is to make temporarily soft links to all files referenced in the source code of a given type (now fixed to .inc) from *incdir* to where the source is being compiled. The restriction to a particular file type is not intrinsic, but just makes easier to remove

the links after compilation without particular bookkeeping. The way to obtain the list of include files uses a number of peculiar Unix tricks and is commented in the **compile** script source.

It has to be noted that include files residing in *sourcedir* (irrespective of their type being .inc or not) can always be accessed. Moreover an include file of type .inc already existing in sourcedir is never overwritten nor deleted (if noclobber is set).

Creating the above links from *incdir* to *sourcedir* may give problems if sourcedir is a link to a non-Unix disk. Therefore in 1.4 all links are done to *relocdir* (this includes an additional link for the source itself), since a cd to there is made before compilation (actually the cd has to be done earlier otherwise the windowsystem command fails if done while the cwd is a link).

4.5 ptlink

VMS version is a `.COM` file. Uses some of the same tricks mentioned above for compile. For the rest should be quite simple. So far no "extra" options (other than those set up in the `.COM` file) can be passed to the linker. However, to support multiple search irrespective of the order, a temporary LNK$LIBRARY is set up before calling the linker.

Unix version is a csh script. Uses similar tricks as compile. Note the different handling of library options, and the fact `f77` is used to call `ld` (see user section above). Note also the linker does not produce any message if there are unresolved references, just an error status. It is possible to pass options to the linker, setting them up with **options** as compiler options (this is the standard behaviour of f77). Of these the -M option (on Sun) or -WM, -Wm (on Ultrix) are useful to produce a load map.

4.6 comlink

Self-explanatory `.COM` file or csh script. Note all the business about purging, and making the executable image "known" is VMS-specific.

In Unix the executable file is by definition executable (!), that is, has execution permission set. Of course it can be invoked with its full path, or just by program name if the *targetdir* is in the current path.

4.6.1 makeknown

In VMS there is nothing like the "path" concept of Unix (or MS-DOS). Also an executable image started with the `RUN` command is not able to retrieve parameters. If one wants to be able to call a program by name, one shall set it up as a foreign task (there are also other ways, but they require system privileges) with a command like :

symbol == **"$***path:progname***.EXE"**

What **makeknown** does is exactly that. Only additionally it checks that the symbol (which is by definition the same as the program name) is not already defined to a different thing. Programmers should note the complex sequence of single and double quotes necessary to verify whether a symbol exists, and to retrieve its value in a variable.

makeknown is designed to be called by **comlink** only. It is possible to call it directly using two arguments (see above, and .COM file listing), but the path to the directory where the executable image is to be found can be expressed only in full VMS format (device and directory) and not in the Uniq format.

As an additional facility, it is possible in VMS to emulate a true path à la Unix (or à la DOS) defining a logical **$PATH** pointing to a comma separated list of directories (separate undocumented utilities **PATH** and **REHASH** are available to manage that). If such a logical is defined, and the directory passed to **makeknown** is in the list, the symbol is defined pointing to **$PATH**:*program* (so that the order in the path decides which executable is actually used).

4.7 complib

Again a `.COM` file and a `csh` script, with some of the usual tricks noted above.

In VMS a `LIBRARY/CREATE` is explicitly requested. Also the "member inserted" messages are generated by `LIBRARY/LOG`. The lexical `F$ENVIRONMENT("DEPTH")` is used to test interactive mode (as defined in user section).

In Unix I had serious problem to test the content of the *libsourcedir* directory using `*.f` as argument and I had to disable it. It is not necessary to create the library (`ar` will do it), but the "member added" message shall be generated by `complib`. The test of "interactive mode" is done via an external source file.

This is called `ifinteractive`, and returns a shell variable flag. The way to test interactivity is the following :

> if the name of the current process (as seen by a "long" `ps`) contains -`bin/csh` (csh), it is interactive (this occurs in a shell or in something sourced from a shell)
>
> else the same trick is attempted on the parent process. If the parent process is another shell script, the name will not appear as above, but as `/bin/csh` *`scriptname`* (in this case we are "not interactive")

In both cases the test on the second word of the name shall be on "`(csh`", not on "`(csh)`" (do not understand why, but it is like that).

4.8 edfor, edinc, edlib

These are three very simple `.COM` files or `csh` scripts, all very similar and self explanatory. Note that `edinc` at the moment requires a file type to be specified, but code to bypass this (if a convention about a standard type for include files is agreed) is already present in comments.

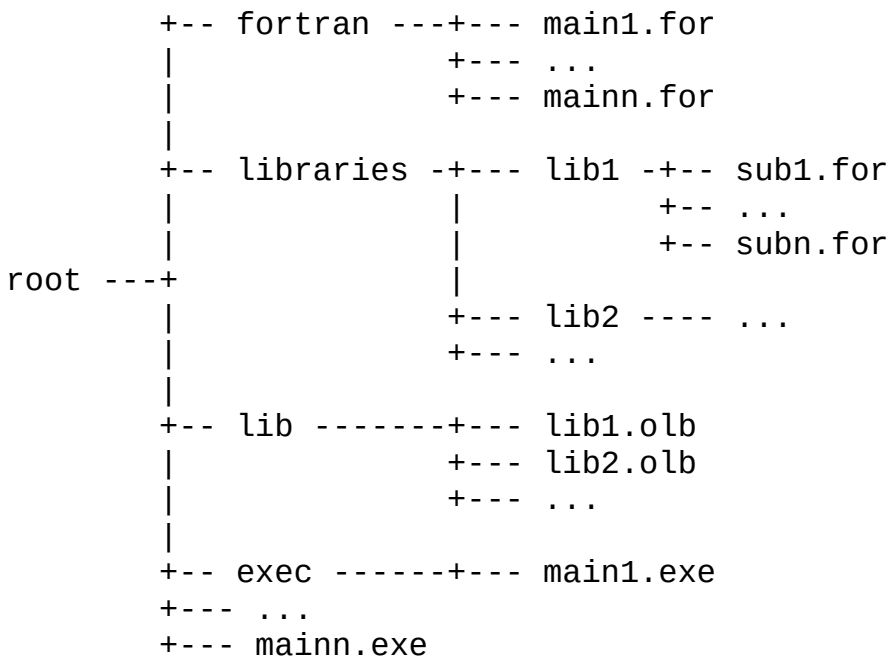
The VMS versions calls the user-defined `EDIT` with the usual trick of redefining `SYS$INPUT` in user mode (normal way of running interactive programs within a `.COM` file).

The Unix version calls the editor using the command "`edit`". This is not normally there, and should be set-up by the user to his favourite editor (the local arrangement is to have this definition as part of the Uniq interface : `edit` is therefore defined on Suns equal to `textedit`, and runs equally well under SunView or OpenWindows, and on DECstation equal to `dxnotepad`).

APPENDIX : examples of directory trees

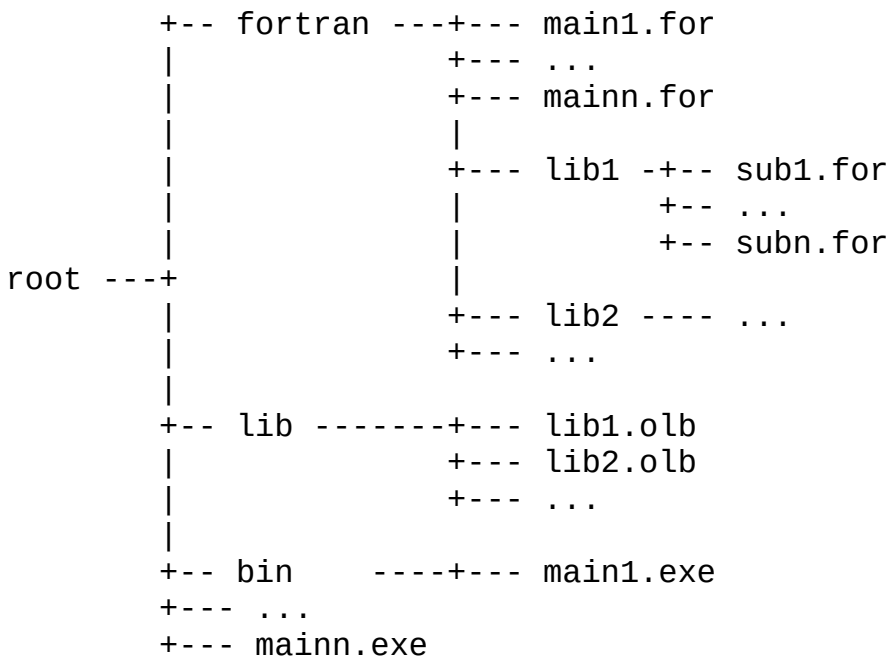
The examples are shown with VMS naming for the files. The role of the varous directories shall be obvious. The relocdir and incdir are not shown.

a) full hierarchy



In this case sourcedir=fortran, libsourcedir=libraries, libdir=lib, targetdir=exec.

b) all sources together



In this case sourcedir=libsourcedir=fortran, libdir=lib, targetdir=bin.

c) all under same root

```
        +--- main1.for
        +--- ...
        +--- mainn.for
        |
        +--- lib1 -+-- sub1.for
        |          +-- ...
        |          +-- subn.for
pkge ---+
        +--- lib2 ---- ...
        +--- ...
        |
        +--- lib1.olb
        +--- lib2.olb
        +--- ...
        |
        +--- main1.exe
        +--- ...
        +--- mainn.exe
```

In this case sourcedir=libsourcedir=libdir=targetdir=pkge.

In all cases remember :

compile	reads a source from *sourcedir*
	writes an object to *relocdir*
ptlink	reads an object from *relocdir*
	writes an executable to *targetdir*
comlink	executes in order `compile` and `ptlink` and
	deletes the object from *relocdir*
complib	reads a source from *libsourcedir.library*
	updates a library in *libdir*
	using as temporary area *relocdir*