FCM Detailed Design

Last updated: 30 March 2007

Met Office
FitzRoy Road, Exeter
Devon, EX1 3PB
United Kingdom

Contents

Introduction

This document contains the Detailed Design for the *Flexible Configuration Management* system which is known as *FCM*.

A lot of information about the design of FCM can be found within:

- The FCM User Guide. This contains all the information which a user needs to know about FCM, including relevant aspects of the design.
- The FCM source code which is, we hope, well commented.

Everything else which you might need to know about the design should be described in the following sections. It includes higher level design information not suited for inclusion in the source code but too detailed to be included in the User Guide. It is intended for those who need to maintain or develop the FCM system or simply for those who wish to obtain a deeper understanding of how the system works. This document consists of the following sections:

Code Management System

The code management commands provided by FCM are generally wrappers around the underlying Subversion commands intended to add functionality or make the system easier to use. Most of the wrappers are very simple and you can find all the information you need from the command reference section of the FCM User Guide and from the source code (which all resides within the Perl package *Fcm::Cm*. This section describes the few areas which merit further discussion.

Interactive Prompts

Several of the commands prompt for further information interactively. This is done using the function "&main::get_input". This allows a different version of the function to be used depending on the calling program. When called by the main `fcm` script this function simply prompts for information on the command line. However, when called from `fcm_gui_internal` (which runs commands within the GUI) this function uses a dialog box generated using Perl-Tk.

This design ensures that the main `fcm` command has no dependency on the Perl *Tk* module.

Merging

Explain the logic used in determining the base path and revision.

GUI

Anything needed here?

Extract System

In this chapter, we shall discuss in detail the design of the extract system. For information of how to use the extract system, please see: FCM System User Guide > The Extract System.

The extract system extracts source from different locations in a version control system (currently only Subversion), combining them with source from the local file system to give a source tree suitable for feeding into the build system. The system is written in a set of Perl modules and has a similar interface to the build system. It shares the same command line interface and many other utilities with the code management system and the build system.

Input and Output

The following inputs are required by the system:

- the location of the destination.
- the location of the source, e.g a URL/revision to one or more branches in the version control system and/or paths in the local file system.
- configurations for the build system.

The system provides the following outputs:

- a source tree suitable for feeding into the build system.
- a configuration file for the build system.
- an expanded version of the current configuration file, so that the current extract can be inherited by a subsequent extract.

Task

To do its job, the extract system executes the following tasks in order:

1. Parse the extract configuration file. There are 2 stages in this process. The file is first read by a primary parser, which process the lines into label/value pairs. The label/value pairs are then parsed in groups (according to the labels) by the extract system, which uses the declaration values to initiate its variables. Inherited extract configuration files are parsed in the same way.
2. Check the destination and the location of any inherited extracts to ensure that they do not contain any lock files. For inherited extracts, ensure that their destinations exist. For a full extract, remove any items created by previous extracts. In any case, set up the destination if it does not exist.
3. Expand declarations for URL keywords and revision keywords
4. Expand declarations for expandable source directories using `svn ls -R`. Discard any directories with no source files in them.
5. Establish the extract priority when dealing with multiple branches. For each directory containing source files, generate a table of specified revision of each branch against the last commit revision of the directory. Discard a directory in a branch if it is unchanged against that of the base. For each directory, the extract priority is established by putting the directory of the base branch at the bottom of the list, followed by the remaining directories for the branches in order of when they are declared. Directories from the local file system always takes precedence, and so are placed at the top of the list.
6. Export each directory in the above list from the version control system into the cache.
7. For each directory, compare the files in each branch according to the priority list. Discard a file in a branch if it has the same content as that of the base. If a file is changed by 2 or more branches, fail the extract unless override mode is specified. In which case, issue a warning and the file in the last declared

branch is used.

8. Copy the file from the cache to the destination if the file does not exist in the destination or if its content differs. Remove a file in the destination if it does not exist in any branch of the cache.
9. Write an expanded extract configuration file.
10. Write of a build configuration file.
11. Mirror the source and configuration files to an alternate location, if necessary.

Build System

In this chapter, we shall discuss in detail the design of the build system. For information of how to use the build system, please see: FCM System User Guide > The Build System.

The build system analyses the directory tree containing a set of source code, processes the configuration, and invokes *make* to compile/build the source code into the project executables. The system is written in a set of Perl modules. It is designed to work with GNU *make*. It creates the *Makefile* and many other dependent files automcatically. The build system uses a similar interface to the extract system. Its configuration file can be produced through the extract system. It also shares the same command line interface and many other utilities with the code management system and the extract system.

Input and Output

The build system has the following input:

- a directory tree populated with a set of Fortran and C source files, and scripts written in Perl, Python, TCL, PVWave or a Unix shell.
- the location of the root directory of the build.
- the tools/commands (and their options) to be used for the build.
- the locations of previous builds, if the current build is to base on them.
- other build options.

Output from the build system includes:

- the targets of the build and their dependencies.
- other files used by the system to build the targets.

Components

The build system uses the following commands, modules and tools:

Name	Category	Description
fcm	Perl executable	Top level command line interface of the FCM system.
fcm_internal	Perl executable	Command wrapper for the compiler and linker.
Fcm::Build	Perl module	Main class that controls the running of the build system.
Fcm::BuildTask	Perl module	A class that performs various "tasks" (such as pre-process and generate interface) for the build system.
Fcm::CfgFile	Perl module	A class for reading from and writing to configuration files.
Fcm::Compiler	Perl module	A class for wrapping the compiler and linker commands.
Fcm::Config	Perl module	A class that contains the configuration settings shared by all FCM components.
Fcm::SrcFile	Perl module	A class that controls the actions on a source file.
Fcm::SrcPackage	Perl module	A class that deals with the actions on a source directory sub-package.
Fcm::Util	Perl module	A collection of utilities shared by all FCM components.
Ecmwf::Fortran90_stuff	Perl module	A utility originally developed by the ECMWF for generating interface blocks for Fortran 9X source files. Modified for adoptation by the FCM system.
make	Unix utility	The *make* build utility. FCM is designed to work with the GNU version of *make*.
sh	Unix shell	The following shell commands are used: "cp", "rm", "mv", "cd" and "touch".
f90aib	Fortran utility	Formerly used by the GEN system as the generator for Fortran 9X interface blocks. It is a freeware developed by Michel Olagnon at the French Research Institute for Exploitation of the Sea. Its use is still supported by FCM, but the ECMWF interface generator is now preferred.

The Build Command

There are several options that can be supplied to the build command. These options are implemented as follows:

- **Archive mode**: the build system generates many files, which are placed in various sub-directories according to their categories. Most of these files are not used in the runtime environment. To prevent a clutter up of disk space, it may be desirable to archive these sub-directories. The system implements this by using the "tar" command to archive each sub-directory. For an incremental build in the same directory, the archived "tar" files are extracted back into the sub-directories so that they can be re-used. It is worth noting that the archive mode should not be used if a build is going to be re-used as a

pre-compiled build, as it will not be possible to extract the "tar" archives back to their original forms. The default is not to archive sub-directories.

- **Full mode**: on an incremental build, the system removes all sub-directories created by the previous build, so that a new build can be performed. The default is to perform an incremental build.
- **Parallel jobs**: this is implemented via the option in GNU *make*. The generated *Makefile* is written with parallel *make* in mind. The default is to perform a serial build.
- **Stage**: this is implemented by running the build system up to and including a named or a numbered stage. The default is to go through all the stages.
- **Targets**: if targets are specified in the build command, overrides the default targets supplied to the *make* command. The default is to build the "all" target.
- **Verbose**: the verbose setting is a variable in the Fcm::Config module. If the option is specified in the build command, it sets this variable to the value supplied to the option. At various places in the build process, it may be useful to print out diagnostic information. This variable contols whether the information will get printed.

The Central/User Configuration File

When we invoke the FCM command, it creates a new instance of Fcm::Config, which reads, processes and stores information from the central and user configuration file. The default settings in Fcm::Config is overwritten by the information provided by the central configuration file. If a user configuration file is found, its settings will take overall precedence. These settings are stored in the Fcm::Config instance, which are parsed to all other modules used by the build system. By convention, the reference to the Fcm::Config instance can normally be fetched by the "config" method for all OO "Fcm::" modules.

The Build Configuration File

When we invoke the build command, it creates a new instance of Fcm::Build, which automatically creates a new instance of Fcm::CfgFile. If an argument is specified in the build command, it is used as the build configuration file if it is a regular file. Otherwise, it is used to search for the build configuration file. If no argument is specified, the current working directory is searched. Fcm::CfgFile will attempt to locate a file called "bld.cfg" under this directory. If such a file is not found, it will attempt to locate it under "cfg/bld.cfg".

Once a file is located, Fcm::CfgFile will attempt to parse it. This is done by reading and processing each line of the configuration file into separate label, value and comment fields. Each line is then pushed into an array that can be fetched using the "lines" method of the Fcm::CfgFile instance. Internally, each line is recorded as a reference to a hash table with the following keys:

- LABEL: the label of a declaration.
- VALUE: the value of a declaration.
- COMMENT: the comment following a declaration or the comment in a comment line.
- NUMBER: the line number of the current line in the source file.
- SRC: the name of the source file.

The information given by each line is "deciphered" by Fcm::Build. The information is processed in the following ways:

- The configuration file type "CFG::TYPE" and version "CFG::VERSION" declarations are stored as properties of the Fcm::CfgFile instance. Fcm::Build uses the information to ensure that it is reading a build configuration file.
- Build target(s) declarations "TARGET" are delimited by space characters. Each target is pushed into the array reference "TARGET".
- The location of the build root directory "DIR::ROOT" and its sub-directories are stored in the "DIR" hash reference in the Fcm::Build instance. The keys of the hash table are the internal names of the sub-directories, and the values are the locations of the sub-directories.

- The source directories "SRC::<pcks>" are stored in the "SRCDIR" hash reference under the Fcm::Build instance. The keys of the hash table are the names of the sub-packages. (If package names are delimited by the double colons "::", they are turned into the double underscores "__" at this stage.) The values of the "SRCDIR" hash reference are new instances of Fcm::SrcPackage. When initialised, Each Fcm::SrcPackage creates a list of source files in its source directory. For each source file in the list, a new instance of Fcm::SrcFile is created, and pushed into the the "SRCFILE" array reference under the Fcm::SrcPackage instance. When initialised, each Fcm::SrcFile instance will attempt to determine the type associated with the source file. This information can be fetched using the "type" method of the Fcm::SrcFile instance.
- Pre-processor switches "PP[::<pcks>]" declarations are stored in the "PP" hash reference under the Fcm::Build instance. The keys of this hash table are the names of the sub-packages, prefix with PP and a pair of underscores, i.e. "PP__<pcks>". The declaration to switch on/off pre-processing globally is stored using the key "PP". The values in the hash table is either true (1) or false (0).
- A tool declaration "TOOL::<name>[::<pcks>]" is stored as a setting in Fcm::Config as ("TOOL", <name>[__<pcks>]. The value of the declaration can then be fetched using the "setting" method of the Fcm::Config instance.
- An exclude dependency declaration "EXCL_DEP[::<pcks>]" is stored as a setting in Fcm::Config as ("EXCL_DEP", <value>, <pcks>, where <value> is the value of the declaration. For an exclude dependency declaration that applies globally, <pcks> is set to a null string "".) The value of an exclude dependency setting is always set to 1.
- Input file extension declaration "INFILE_EXT::<ext>" is stored as a setting in Fcm::Config as ("INFILE_EXT", <ext>). The value of the setting can then be fetched using the "setting" method of the Fcm::Config instance.
- Output file extension declaration "OUTFILE_EXT::<type>" is stored as a in Fcm::Config as ("OUTFILE_EXT", <type>). The value of the setting can then be fetched using the "setting" method of the Fcm::Config instance.
- For each "USE" declaration to use a pre-compiled build, a new instance of Fcm::Build is created. The instance Fcm::Build for the previous build creates a new instance of Fcm::CfgFile for its configuration file. The configuration of the previous build is read and processed as described above. The current instance of Fcm::Build will then attempt to inherit the settings of the previous build where appropriate. The Fcm::Build instance for the pre-compiled build is pushed into the "USE" array reference under the current Fcm::Build.
- The inherit flags for targets "INHERIT::TARGET", source directories "INHERIT::SRC[::<pcks>]" and pre-processor switches "INHERIT::PP[::<pcks>]" are stored in the "INHERIT" hash reference under the Fcm::Build instance. For declarations that apply globally, the keys to the hash table are "TARGET", "SRCDIR" or "PP". If the declarations apply to <pcks>, the keys are "SRCDIR__<pcks>" or "PP__<pcks>". (The inherit target flag always applies globally.)

Unless the search source flag "SEARCH_SRC" is switched off (0) in the build configuration, Fcm::Build will attempt to search the source sub-directory "src/" of the build root recursively for source directory sub-packages. The source directories obtained in the search are treated as if they are declared using "SRC::<pcks>" in the build configuration file.

Compiler Flags

As discussed in the user guide, if you declare the Fortran compiler flags without specifying a sub-package, the declaration applies globally. Otherwise, it only applies to the Fortran source files within the sub-package. This is implemented via a simple "tool selection" mechanism. You may have noticed that all TOOL declarations (and TOOL settings in Fcm::Config) are turned into an environemnt variable declaration in the generated *Makefile*. For example, if we have a "FFLAGS__bar__egg__ham__foo" declaration, it will be declared as an environment variable in the generated *Makefile*. Suppose we have a source file "foo.f90" under the sub-package "bar::egg::ham". When we invoke the compiler wrapper (i.e. "fcm_internal" and "Fcm::Compile") to compile the source file, the system will first attempt to select from the FFLAGS environment variable that matches the sub-package of the source file, which is "FFLAGS__bar__egg__ham__foo" in this case. If the environment variable does not exist, it will attempt to go one sub-package up, i.e. "FFLAGS__bar__egg__ham", and so on until it reaches the global "FFLAGS"

declaration, (which should always exists).

For changes in compiler flags declaration, the build system should trigger re-compilation of required targets only. This is implemented using a "flags" file system. These "flags" files are dummy files created in the "flags/" sub-directory of the build root. They are updated by the "touch" command. The following dependencies are followed:

- Source files are dependent on its own "flags" file. E.g. the file Ops_Switch in sub-package ":ops::code::OpsMod_Control" is dependent on "FFLAGS__ops__code__OpsMod_Control__Ops_Switch.flags".
- The "flags" file of a source file is dependent on the "flags" file of its container sub-package. E.g. the above flags file is dependent on "FFLAGS__ops__code__OpsMod_Control.flags".
- The "flags" file of a sub-package is dependent on the "flags" file of its container sub-package. E.g. the above is dependent on "FFLAGS__ops__code.flags", which is dependent on "FFLAGS__ops.flags".
- The "flags" file of a top-level package is dependent on the "flags" file of the global flags. E.g. "FFLAGS__ops.flags" is dependent on "FFLAGS.flags".
- The "flags" file of the global "flags" file is dependent on the "flags" file of the compiler command. E.g. "FFLAGS.flags" is dependent on "FC.flags".

The system records changes in declared tools using a cache file, (called ".bld_tool", located at the ".cache/" sub-directory of the built root). It is basically a list of "TOOL::" declarations for the latest build. When an incremental build is invoked, the list is compared against the current set. If there are changes (modification, addition and deletion) in any declarations, the timestamp of the corresponding "flags" files will be updated. Files depending on the updated "flags" file will then be considered out of date by *make*, triggering a re-build of those files.

Fortran 9X Interface Block Generator

The build system generates an interface block file for each Fortran 9X source file. If the original source file has been pre-processed, the system uses the pre-processed source file. Otherwise, the system uses the original source file. For each source file containing standalone subroutines and functions, the system will generate an interface file containing the interfaces for the subroutines and functions. The interface files for other Fortran 9X source files are empty.

Fcm::Build controls the creation of interface files by searching for a list of Fcm::SrcFile instances containing Fortran 9X source files, by calling the "is_type ('FORTRAN9X')" method of each Fcm::SrcFile instance. For each of Fortran 9X source file, a Fcm::BuildTask is created to "build" the interface file. The build task is dependent on the interface generator. The interface files will be re-generated if we change the interface generator. The generated interface is held in an array initially. If an old file exists, it is read into an array so that it can be compared with the current one. The current interface is written to the interface file if it is not the same as the old one, or if an old one does not already exist.

FCM supports the use of *f90aib* and the ECMWF interface generator. The latter is the default.

Depdendency Scanner

For each source directory sub-package, the build system scans its source files for dependency information. The dependency scanner uses a pre-defined set of patterns and rules in Fcm::Config to determine whether a line in a source file contains a dependency. Only source files of supported types are scanned. The dependency information of a sub-package is stored in the memory as well as a cache file. The latter can be re-used by subsequent incremental builds. In an incremntal build, only those source files newer than the cache file is re-scanned for dependency. The cache file is read/written using temporary instances of Fcm::CfgFile.

The control of the source file selection process is handled by the Fcm::SrcPackage instances, while the actual dependency scans are performed via the scan_dependency method of the Fcm::SrcFile instances.

A dependency has a type. For example, it can be a Fortran module or an include file. The type of a dependency determines how it will be used by the source file during the *make* stage, and so it affects how the *make* rule will be written for the source file. In memory, the dependency information is stored in a hash table, which can be retrieved as a property of the Fcm::SrcFile instance. The keys of the hash table are the dependency items, and the values are their types.

A dependency is not added to the hash table if it matches with an exclude dependency declaration for the current sub-package.

While the dependency scanner is scanning through each line of a Fortran source file, the system also attempt to determine its internal name. This is normally the name of the first compilable program unit defined in the Fortran source file. The internal name is converted into lowercase (bearing in mind that Fortran is case insensitive), and will be used to name the compiled object file of the source file.

The package configuration file is a system to bypass the automatic dependency scanner. It can also be used to add extra dependencies to a source file in the package. The configuration file is a special file in a source package. The lines in the file is read using a temporary instance of Fcm::CfgFile created by Fcm::SrcPackage. All declarations in a package configuration file apply to named source files. The declarations set the properties of the Fcm::SrcFile instance associated with the source file. It can be used to add dependencies to a source file, and to tell the system to bypass automatic dependency scanning of the source file. Other modifications such as the internal name (object file name) of a source file, or the target name of the executable can also be set using the package configuration file in the package containing the source file.

Make Rule Generator

The dependency information is used to create the *Makefile* fragments for the source directory sub-packages. A *Makefile* fragment is updated if it is older than its corresponding dependency cache file.

The following is a list of file types and their *make* rule targets:

File type		Targets
SOURCE	**all**	• compile: object file • touch: flags file for compiler flags
	FPP and C	If the original source has not been pre-processed: • touch: flags file for pre-processor definition macros
	PROGRAM	• load: executable binary file • touch: flags file for loader (linker) flags
	all except PROGRAM	• touch: "done" file to denote the resolution of all external objects.
	all FORTRAN except PROGRAM and MODULE	• interface: "interface" file to denote that all dependent module information files are up to date.
INCLUDE		• cp: "include" file to "inc/" sub-directory • touch: "idone" file to denote the resolution of all external objects.
EXE and SCRIPT		• cp: executable file to "bin/" sub-directory
LIB		• ar: archive object library file

The resulting *Makefile* is made up of a top level *Makefile* and a list of include ".mk" files, (one for each sub-package). The toplevel *Makefile* consists of useful environment variables, including the search path of each sub-directory, the build tools, the verbose mode and the VPATH directives for different file types. It has two top level build targets, "all" and "clean". The "all" target is the default target, and the "clean" target is for removing the previous build from the current build root. It also has a list of targets for building the top level and the container sub-package "flags" files. At the end of the file is a list of "include" statements to include the ".mk" files.

A the top of each of ".mk" files are local variables for locating the source directories of the sub-package. Below that are the rules for building source files in the sub-package.

Pre-processing

As discussed in the user guide, the PP switch can be used to switch on pre-processing. The PP switch can be specified globally or for individual sub-packages. (However, it does not go down to the level of individual source files.) The "inheritance" relationship is similar to that of the compiler flags.

Currently, only Fortran source files with uppercase file extensions and C source files are considered to be source files requiring pre-processing. If a sub-package source directory contains such files and has its PP switch set to ON, the system will attempt to pre-process these files.

The system allows header files to be located anywhere in the source tree. Therefore, a dependency scan is performed on all files requiring pre-processing as well as all header files to obtain a list of "#include" header file dependencies. For each header file or source file requiring pre-processing, a new instance of Fcm::BuildTask is created to represent a "target". Similar to the logic in *make*, a "target" is only up to date if all its dependencies are up to date. The Fcm::BuildTask instance uses this logic to pre-process its files. Dependent header files are updated by copying them to the "inc/" sub-directory of the build root. The "inc/" sub-directory is automatically placed in the search path of the pre-processor command, usually by the "-I" option. Pre-processing is performed by a method of the Fcm::SrcFile instance. The method builds the command by selecting the correct set of pre-processor definition macros and pre-processor flags, using an inheritance relationship similar to that used by the compiler flags. Unlike *make*, however, Fcm::BuildTask

only updates the target if both the timestamp and the content are out of date. Therefore, if the target already exists, the pre-processing command is only invoked if the timestamp of the target is out of date. The output from the pre-processor will then be compared with the content in the target. The target is only updated if the content has changed.

Once a source file is pre-processed, subsequent build system operations such as Fortran 9X interface block generation and dependency scan (for creating the *Makefile*) will be based on the pre-processed source, and not the original source file. If a source file requires pre-processing and is not pre-processed at the pre-processing stage, it will be left to the compiler to perform the task.

File Type Register

The build system file type register is a simple interface for modifying the default settings in the Fcm::Config module. There are two registers, one for output file type and one for input file type.

The output file register is the simpler of the two. It is implemented as a hash table, with the keys being the names of the file types as known by the system internally, and the values being the suffices that will be added to those output files. Therefore, the output file register allows us to modify the suffix added to an output file of a particular type.

The input file register allows us to modify the type flags of a file named with a particular extension. The type flags are keywords used by the build system to determine what type of input files it is dealing with. It is implemented as a list of uppercase keywords delimited by a pair of colons. The order of the keywords in the string is insignificant. Internally, the build system determines the action on a file by looking at whether it belongs to a type that is tied with that particular action. For example, a file that has the keyword "SOURCE" in its type flag will be treated as a compilable source file, and so it will be written to the *Makefile* with a rule to compile it.

The *Makefile*

The following items are automatically written to the *Makefile*:

- The root directory and sub-directories of the current build, as environment variables FCM_ROOTDIR, etc.
- The search path of the different types of output file, using a set of "vpath" directives.
- TOOL declarations in the build configuration file are exported as environment variables.
- The diagnostic verbose mode information.
- The default build targets.
- The rules for building the targets for all the source files.

The Compiler/Linker Wrapper

Compile and link are handled by the *fcm_internal* wrapper script. The wrapper script uses the environment variables exported by the *Makefile* to generate the correct compiler (or linker) command for the current source (or object) file. Depending on the diagnistic verbose level, it also prints out various amount of diagnostic output.

For compilation, the wrapper does the following:

1. Select the correct compiler for the current source file.
2. Specify the output file name for the current source file.
3. If pre-processing is left to the compiler, specify the definition macros, if any, for the pre-processor.
4. Specify the "include" path, in case the source file has dependencies on include files.
5. Specify the "compile only" option, if it is not already set.

6. Add any other user defined flags to the compiler command.
7. Run the command, sending the output to a temporary directory.
8. If the compile succeeded, move the output from the temporary directory to the output object directory.
9. Otherwise, delete the output from the temporary directory.
10. If there are Fortran module definition files (*.mod, *.MOD, etc), move them to the "inc/" sub-directory.

For linking, the wrapper does the following:

1. Create a temporary object archive library with all the object files currently residing in the output object sub-directory.
2. Select the correct linker for the current main program object file.
3. Specify the output file name for the main program.
4. Specify the main program object file.
5. Specify the link library search path and the temporary link library.
6. Add any other user defined flags to the linker command.
7. Run the command, sending the output to a temporary directory.
8. If the link succeeded, move the output from the temporary directory to the "bin/" sub-directory of the build root.
9. Otherwise, delete the output from the temporary directory.
10. Remove the temporary object archive library.

Inheriting from a Pre-compiled Build

A build can inherit configurations, source files and other items from a previous build. At the Perl source code level, this is implemented via hash tables and search paths. At the *Makefile* level, this is implemented using the "vpath" directives. The following is a summary:

- Build targets are stored in a list. Targets that are specified in the configuration file in the previous build are pushed into the list first. A target that is specified in the current configuration file is pushed into the list if it has not already been specified.
- Each source directory has a unique sub-package identifier and a path location. If a source directory in the previous build has the same sub-package identifier, its location as specified in the previous build will be placed in the search path of the source directory. The directory specified in the current build is searched for files before the directory specified in the previous build.
- Tool settings such as compiler flags are set via the configuration hash table. Settings declared in the previous build overrides those of the default, and settings declared in the current build overrides those declared in the previous build.
- Ditto for exclude dependency, input file extension and output file extension declarations.
- Build items such as object files, executable files and other dummy files are inherited via search path (and/or "vpath"). For example, the system searches the "obj/" sub-directory of the current build for object files before looking at that of the previous build. The system does not re-build an item that exists in the previous build and is up to date.