CHAPTER 4

The Makefile, Project, and Workspace Creator (MPC)

4.1 Introduction

Maintaining multiple build tool files for a multi-platform project can be quite a challenge, especially when the project structure and platforms are constantly changing and evolving. A project may support Makefiles, Visual C++ project files, Borland Makefiles, and many others. Adding files, deleting files, changing project options or even changing the name of the target within your project will require you to expend time updating each build tool file. What you need instead is a single location to store project specific information to avoid repetitious, tedious modifications to multiple build tool files. This is where Makefile Project Creator (MPC) comes into the picture.

MPC can be used to generate build tool specific project files from a generic mpc file. The MPC project file is a collection of source files that make up a single build target. MPC uses platform specific input along with mpc files and generates build tool specific files like makefiles, Visual C++ workspace and project files, Visual Studio solution and project files, etc.

MPC provides many advantages over the build tool files it replaces. It provides mechanisms for minimizing maintenance of project build files. It

does this through support for project inheritance and defaults for all aspects of a project, and the syntax is simple and easy to use and maintain. These and other features will be discussed in detail in the following sections. A complete example of the use of MPC is shown in section 4.3.3.8.

4.2 Using MPC

An MPC project is a set of parameters that describe an individual build target (such as a library or executable). These parameters include the target name, include paths, source files, header files, etc. One or more projects can be defined within a single mpc file. An MPC workspace is just an arbitrary collection of projects.

Projects can be generated (without workspaces) by using the mpc.pl script. Multiple mpc files can be passed to this script. If no mpc files are passed to the script, it will search for project-related files (such as source files, header files, etc.) and incorporate them into a default project.

Figure 4-1 shows a high-level view of project file generation using mpc.pl.

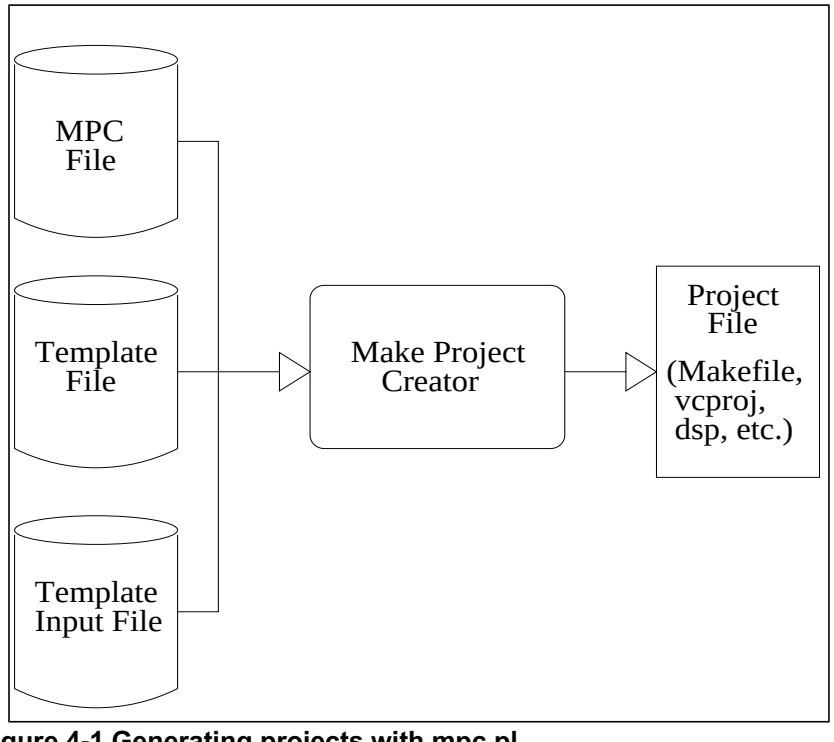

Figure 4-1 Generating projects with mpc.pl

To generate workspaces, you must run `mwc.pl`. This script will generate projects from mpc files and create a workspace based on those mpc files. If no mwc files are passed to the script, it will search in the current directory and its subdirectories for all mpc files and incorporate them into a single workspace.

For make based project types (make, gnuace, bmake, nmake), a workspace is just a top-level makefile. But, for graphical interfaces such as Visual Studio, a workspace is the top-level file that groups all of the project files together.

Figure 4-2 shows a high-level view of workspace file generation using mwc.pl.

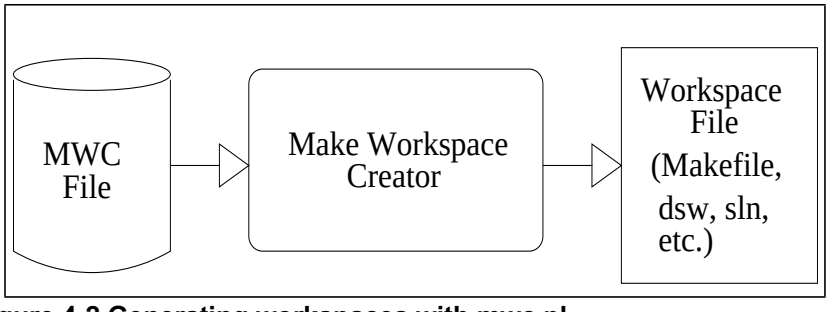

Figure 4-2 Generating workspaces with mwc.pl

4.2.1 Supported Build Tools

MPC generates workspaces and projects for use with many build tools. Table 4-1 lists the MPC types (used with mpc's `-type` option) and their associated build tools.

Table 4-1 MPC Types

Type	Build Tool
`automake`	GNU Automake.
`bmake`	Borland Make.
`cbx`	Support for Borland CBuilderX is incomplete.
`em3`	eMbedded Visual C++ 3.00 and 4.00.
`ghs`	Support for Green Hills C++ Builder is incomplete.
`gnuace`	GNU Make for ACE/TAO only (ACE/TAO extension).
`make`	Generic make. The makefiles generated by this project type can be used with any version of make. However, due to configuration issues, it should not be use with ACE or TAO.
`nmake`	Microsoft NMake.
`sle`	Support for Visual SlickEdit is incomplete.
`vc6`	Visual C++ 6.0.
`vc7`	Visual C++ 7.0.
`vc71`	Visual C++ 7.1.
`vc8`	Visual C++ 8.0.

4.2.2 Command Line

The command line options for the workspace creator (`mwc.pl`) and the project creator (`mpc.pl`) are exactly the same. The project creator is used to generate one or more separate projects by passing mpc files to it on the command line. The workspace creator is used to generate one or more workspaces and the projects related to those workspaces.

Table 4-2 describes each option with the more commonly used options in bold and project specific options in italics.

Table 4-2 Command Line Options

Option	Description
-base	This option allows the user to force any project to inherit from a specified base project. This option can be used multiple times to force multiple inheritance upon a project.
-exclude	If this option is used with `mwc.pl`, the directories or mwc files provided in a comma separated list will be excluded when searching for mpc files. Each element provided for exclusion should be relative to the starting directory. This option has no effect when used with `mpc.pl`.
-expand_vars	This option instructs MPC to perform direct replacement of $() variables with the values from the environment (if the `-use_env` option is used) or the values specified by the `-relative` option.
-feature_file	This option allows the user to override the default feature file (`MPC/config/default.features` or `ACE_wrappers/bin/MakeProjectCreator/config/default.features`) which may or may not exist. This file can be used to override feature values specified in the `global.features` file located in the `config` directory. Feature files are described in section 4.3.2.3.
-features	Specifies the feature list to set before processing. This is a comma separated list and should contain no spaces.
-genins	This option instructs MPC to generate an “install” file after processing each project. These “install” files can be used with the prj_install.pl script which will copy portions of the project related files into a user specified location.
-global	This option specifies the global input file. Values stored within this base project are applied to all generated projects. The default value is `ACE_wrappers/bin/MakeProjectCreator/global.mpb or MPC/config/global.mpb`.

Table 4-2 Command Line Options

Option	Description
-hierarchy	If this option is used with `mwc.pl`, it will generate a workspace at each directory between the directory in which it is run and the location of a processed mpc file. This option has no effect when used with `mpc.pl` and is the default for "make" based workspace types.
-include	Include search directories are added with this option. These search directories are used when locating base projects, template input files and templates. It can be used multiple times on the same command line.
-language	This option is used to specify which language to assume when generating projects. The default language is `cplusplus`, but `csharp`, `java` and `vb` are also supported.
-make_coexistence	Make based project types that normally name the workspace `Makefile` (`bmake` or `nmake`) will name the generated output files such that they can coexist within the same directory. In essence, the `bmake` and `nmake` workspace names will not be `Makefile`, but the name of the workspace followed by the project type (`.bmake` or `.nmake`).
-name_modifier	This option allows the user to modify the output names of projects and workspaces. These are usually determined by either the mpc or mwc file, but can be modified using a pattern replacement. The parameter passed to this option will be used as the pattern and any asterisks (*) found in the pattern will be replaced with the project or workspace name depending on which type of file is being created.
-apply_project	This option is only useful with the `-name_modifier` option. When used in conjunction with `-name_modifier`, the pattern will be applied to the project name in addition to the project or workspace name.
-noreldefs	This option specifies that the default relative definitions should not be generated. See the `-relative` option below.
-notoplevel	This option tells `mwc.pl` to generate all workspace related project files, but do not generate the associated workspace. This option tells `mpc.pl` to process all mpc files passed in, but it will not generate any project files.
-recurse	Search from the current directory for any input files and process them from the directory in which they are located.

Table 4-2 Command Line Options

Option	Description
-relative	Relative paths are used to replace variables enclosed with `$()`. By default, any environment variable that ends in `_ROOT` will be automatically used as a relative path replacement. For more information see “The -relative Option.” on page 40.
-template	This option allows a user to specify an alternate template. Each project type has its own template and this option allows a user to override the default template.
-ti	Each project type has a set of template input files. With this option the default template input file can be overridden for a particular project type. For more information see “The -ti Option.” on page 40.
-type	This option specifies the type of project or workspace to be generated. It can be specified multiple times to generate different project types for a single set of input files.
-use_env	This option instructs MPC to replace all `$()` instances with the corresponding environment variable value instead of using values provided by the -relative option.
-static	Specifies that static project files will be generated from the MPC projects. The default is to generate dynamic project files.
-value_project	Use this option to override an mpc project assignment from the command line. This can be used to introduce new name value pairs to a project. However, it must be a valid project assignment. For more information see “The -value_project Option.” on page 41.
-value_template	This option can be used to override existing template input variable values from the command line. It can not be used to introduce new template input name value pairs. For more information see “The -value_template Option.” on page 41.
-version	The MPC version is printed and no files are processed.
-complete	The previously undocumented `complete` option can be used to generate a tcsh `complete` command that allows a user of the tcsh shell to complete on options as well as file names.

4.2.2.1 Additional Option Descriptions

Some of the options in Table 4-2 require an expanded explanation. You will find more information on the `-relative`, `-ti`, `-value_project` and `-value_template` options below.

The -relative Option.

Some project types do not (completely) support the idea of accessing environment variables through the use of `$()`, and therefore MPC must ensure that generated projects are usable in these cases. In order to avoid the existence of `$()` variables within the generated project files, relative paths are put in place of those (where possible).

The `-relative` option takes a single parameter of a name value pair, for example:

```
mwc.pl -relative PROJ_TOP=/usr/projects/top
```

In above example, if the text `"$(PROJ_TOP)"` is found as a value for any mpb, mpc, mpd, or mpt variable then it is replaced by a path that is relative to `/usr/projects/top`. For example, if an mpc file located under `/usr/projects/top/dir` contained the following:

```
project {
  includes += $(PROJ_TOP)
}
```

The generated project file would contain text similar to:

```
CPPFLAGS += -I..
```

The `$(PROJ_TOP)` string was replaced with a directory value that is relative to the directory in which the mpc file is located.

The -ti Option.

The `-ti` option allows you to identify different template input files based on the type of target being built. Template input files correspond to four different categories: `dll`, `lib`, `dll_exe`, and `lib_exe`. Not all project types distinguish between the different categories, but the templates for various project types will be combined with different template input files, depending on the build target type, to generate different output.

To override the default template input file names, a `-ti` option is provided. The `-ti` option takes a single parameter of the form `type:file`. The `type` is one of the four categories stated above and the `file` is the base name of an mpt file located somewhere in the include search paths.

The following example shows a usage of the -ti option. It says that when generating a static project (lib), use the `vc7lib` template input file and when generating a dynamic project (dll), use the `vc7dll` template input file:

```
mpc.pl -type vc71 -ti lib:vc7dsplib -ti dll:vc7dspdll
```

These happen to be the default values for the `vc71` type, but it illustrates that a different template input can be specified for each category.

The -value_project Option.
The `-value_project` option can be useful when the need arises to modify the value of an MPC variable across one or more mpc files. For example, if you wanted to generate all of your projects with an additional include search path you would run the following command:

```
mwc.pl -value_project includes+=/include/path
```

In the above example, an additional include search path of `/include/path` would be placed in all generated projects.

The -value_template Option.
This option modifies existing or adds new template input name/value pairs. For example, if you wanted to generate dynamic `vc71` projects with only `Release` targets, you would run the following command:

```
mwc.pl -type vc71 -value_template configurations=Release
```

To find out what template input variables are defined, see the individual mpd file of interest (`$ACE_ROOT/bin/MakeProjectCreator/templates/*.mpd` and `$MPC_ROOT/templates/*.mpd`) and search for names used within <% and %>. Names that are not listed as project keywords (Table 4-3 on page 46) are template variables.

4.2.3 Environment Variables

MPC recognizes a few environment variables that alter the way it performs certain tasks. The sections below describe each one and the effect it has on MPC.

MPC will use the options defined in `MPC_COMMANDLINE` as if they were given on the command line to mwc.pl or mpc.pl. The environment value will be prepended to options actually passed to mwc.pl or mpc.pl on the actual command line.

The `MPC_DEPENDENCY_COMBINED_STATIC_LIBRARY` environment variable only affects the way workspace dependencies are created for *static* projects with the `em3`, `vc6`, `vc7`, `vc71` and `vc8` project types. If this environment variable is set, MPC will generate inter-project dependencies for libraries within a single workspace. This is usually not desired since adding these dependencies in a static workspace has the side effect of including dependee libraries into the dependent library.

If the `MPC_LOGGING` environment variable is set, MPC will parse the value and provide informational, warning and diagnostic messages depending on it's setting. If the value contains `info=1`, informational messages will be printed. If it contains `warn=1`, warning messages will be printed. If it contains `diag=1`, diagnostic messages will be printed. And lastly, if it contains `detail=1`, detail messages will be printed. If it contains none of these, MPC will act as if `MPC_SILENT` was set.

The `MPC_SILENT` environment variable instructs MPC not to print any messages, except error messages. The progress indicator is still printed.

If `MPC_VERBOSE_ORDERING` is set, MPC will warn the user about references to projects in the "`after`" keyword that have not been processed. This only has an effect when running `mwc.pl`.

4.3 Writing MPC and MWC Files

You may want to familiarize yourself with the various input files for MPC. The input file types and the syntax of each are discussed in the sections below.

4.3.1 Input Files

There are four different input files associated with MPC. For most users of MPC, the main files of concern are mpc and mwc files.

4.3.1.1 Project Files (mpc)

Project files, those with the `mpc` extension, contain such things as include paths, library paths, source files and inter-project dependencies. An mpc file can contain one or more "projects" each of which needs to be uniquely named to avoid project generation errors. Projects represent build targets such as libraries and executables.

4.3.1.2 Workspace Files (mwc)

Workspaces are defined by providing a list of mpc files, directories or other mwc files in a single mwc file. For each mpc file, the Workspace Creator calls upon the Project Creator to generate the project. After all of the projects are successfully generated, the tool-specific workspace is generated containing the projects and any defined inter-project dependency information (if supported by the build tool). An mwc file can contain one or more "workspaces," each of which needs to be uniquely named. If no workspace files are provided to the workspace creator, the current directory is traversed and any mpc files located will be part of the workspace that is generated.

4.3.1.3 Base Project Files (mpb)

One of the many unique and useful features of MPC is that the project definition files can use inheritance. Project inheritance allows a user to set up a base project (mpb file) that can contain information that is applicable to all derived projects. Common project attributes, such as include paths, library paths, and inter-project dependencies, could be described in this base project and any project that inherits from it would contain this information as well.

4.3.1.4 Base Workspace Files (mwb)

As with projects, workspaces can also inherit from other workspaces. A base workspace can provide workspace information that may be common to other workspaces.

4.3.2 General Input File Syntax

In this section we discuss the syntax of the various files. We also describe some of the default values that go along with these files.

4.3.2.1 mwc and mwb

Workspaces can contain individual mpc files or directories. There can be one or more workspaces defined within a single mwc file.

```
workspace(optional name): optional_base_workspace {
   file.mpc
   directory
   other.mwc

  exclude(vc6, vc7, vc71, vc8, nmake) {
    this_directory
  }
}
```

A workspace can be given a name. This is the value given in the parentheses after the keyword `workspace`. If the workspace is not given a name, the workspace name is taken from the name of the mwc file without the extension.

Workspaces can also inherit from other workspaces. In the above example, `optional_base_workspace` would be the base name of an mwb file with no extension that contains workspace information. This information would then be included in each workspace that inherits from it.

The lines between the curly braces contain assignments, mpc files, directories, other workspace files or exclusion sections. The mpc files listed will be included in the workspace. If a directory is listed within the workspace, the workspace creator will recursively traverse that directory and use any mpc files that are found. If a workspace file is listed it will be aggregated into the main workspace.

A workspace can have assignments interspersed within the directories and mpc files. These assignments modify the way projects are generated.

The `cmdline` setting can be used to provide command line options that would normally be passed to `mwc.pl` (see Table 4-2). However, the `-type`, `-recurse`, `-noreldefs`, `-make_coexistence`, `-genins`, `-into` and `-language` options as well as input files are ignored. Environment variables may be accessed through `$NAME`, where `NAME` is the environment variable name. The `cmdline` assignment may be useful for workspaces that require specific `mwc.pl` options in order to process correctly.

The only other setting supported by mwc.pl is `implicit`. If `implicit` is set to `1` then default project files are generated in each directory where no mpc file exists. The `implicit` keyword can also be set to the name of a base

project. In this case, the implicitly generated project will inherit from the base project specified in the assignment. Either way, if the directory does not contain files that can be used within a project, no project is created. Setting `implicit` can be useful when you want to define specific workspaces, but the MPC defaults are sufficient for the directories involved within the workspace.

Scoped assignments are assignments that are associated with specific mpc files or directories listed with the scope of the assignment. The following example shows a scoped assignment of `cmdline` that only applies to `one of the mpc files listed in the workspace`. In this example, `directory/foo.mpc` would be processed as if the `-static` option had been passed on the command line whereas other directories and mpc files would not.

```
workspace {
   ...
   static {
      cmdline += -static
      directory/foo.mpc
   }
   exclude(gnuace, make) {
      some.mpc
   }
}
```

Exclusion sections are used to prevent directories and mpc files from being processed. These excluded directories and mpc files will be skipped when generating project files and workspaces. The `exclude` keyword accepts project types within the parentheses (as above), which will cause the workspace creator to only exclude the listing for particular types. If no types are provided, exclusion will take place for all project types.

Comments are similar to the C++ style comments. Any text after a double slash (`//`) is considered a comment.

4.3.2.2 mpc and mpb

Project Declarations

Project declarations are similar to workspace declarations, but are a bit more complex. An mpc file can contain one or more “projects” and each project can inherit from base projects.

```
project(optional name): base_project, another_base_project {
  exename   = client
  includes += directory_name other_directory
  libpaths += /usr/X11R6/lib

  Header_Files {
    file1.h
    file2.h
    fileN.h
  }

  Source_Files {
    file1.cpp
    file2.cpp
    fileN.cpp
  }
}
```

If the optional project name is not given, then the project name is taken from the name of the mpc file without the extension. Therefore, if your mpc file is going to contain multiple projects, it is important to provide project names to prevent each generated project from overwriting the other. MPC will issue an error and stop if duplicate project names are detected.

Base Projects

Base projects can be of the extension `mpb` and `mpc`. If a file with the name of the base project with an `mpb` or `mpc` extension cannot be found within the mpc include search path, a fatal error is issued and processing halts.

Assignment Keywords

Table 4-3 shows the keywords that can be used in an assignment (i.e. `=`, `+=` or `-=`) within an mpc file. The most commonly used keywords are shown in bold face.

Table 4-3 Assignment Keywords

Keyword	Description
after	Specifies that this project must be built after 1 or more project names listed.
avoids	Specifies which features should be disabled in order to generate the project file. Under the GNUACE type, it also specifies which make macros should not be set to build the target.

Table 4-3 Assignment Keywords

Keyword	Description
custom_only	This setting instructs MPC to create projects that only contain custom generation targets. Any files included in the projects will be provided by custom component lists defined through the use of `Define_Custom`.
dynamicflags	Specifies preprocessor flags passed to the compiler when building a dynamic library.
dllout	If defined, specifies where the dynamic libraries will be placed. This overrides libout in the dynamic case.
exename	Determines that the project will be an executable and the name of the executable target.
includes	Specifies one or more directories to supply to the compiler for use as include search paths.
install	Specifies where executables will be installed.
libout	Specifies where the dynamic and static libraries will be placed.
libpaths	Specifies one or more directories to supply to the compiler for use as library search paths.
libs	Specifies one or more libraries to link into the target. Library modifiers may be added when being processed in the template file. For example, library modifiers are added when using the vc6 project type.
lit_libs	This is the same as libs except that a library modifier will not be added.
macros	Values supplied here will be passed directly to the compiler as command line defined macros.
pch_header	The name of the precompiled header file. See the discussion below this table for more information.
pch_source	The name of the precompiled source file. See the discussion below this table for more information.
pure_libs	This is similar to `lit_libs` except that no prefix or extension is added to the names specified.
postbuild	If this is defined in the project, the value will be interpreted as commands to run after the project has been successfully built. The <% %> construct (See “Template Files (mpd)” on page 66.) can be used within this value to access template variables and functions of the template parser.

Table 4-3 Assignment Keywords

Keyword	Description
recurse	If set to 1, MPC will recurse into directories listed under component listings (such as Source_Files, Header_Files, etc.) and add any component corresponding files to the list. This keyword can be used as a global project setting or a component scoped setting.
requires	Specifies which features should be enabled in order to generate the project file. Under the GNUACE type, it also specifies which make macros should be set to build the target.
sharedname	Determines that the project will be a library and the name of the dynamic library target. See the discussion below this table for more information.
staticflags	Specifies preprocessor flags passed to the compiler when building a static library.
staticname	Determines that the project will be a library and the name of the static library target.
tagchecks	For GNUACE Make only, specifies one or more names to search for in the macros specified by tagname.
tagname	Specifies the GNUACE Make macro to check before building the target.
version	Specifies the version number for the library or executable.

Assignments can also use the `+=` and `-=` operators to add and subtract values from keyword values.

If a `sharedname` is specified in the mpc file and `staticname` is not used, then `staticname` is assumed to be the same as `sharedname`. This also applies in the opposite direction.

If neither exename, sharedname nor staticname is specified, MPC will search the source files for a `main` function. If a `main` is found, the exename will be set to the name of the file, minus the extension, that contained the `main` function. Otherwise, sharedname and staticname will be set to the project name.

If the project name, exename, sharedname or staticname contain an asterisk it instructs MPC to dynamically determine a portion of the name based on certain defaults. If the project name contains an asterisk, then the asterisk will be replaced with the default project name. If exename, sharedname or

staticname contains an asterisk, then the asterisk will be replaced with the project name.

If the `pch_header` keyword is not used and a file exists, in the directory in which the mpc file is located, that matches `*_pch.h` it is assumed to be the precompiled header for that directory. If there are multiple pch files in the directory, then the precompiled header that closely matches the project name will be chosen. Similar logic applies for the `pch_source` keyword.

Components

An mpc file can also specify the files to be included in the generated "project" file. These files are specified using the component names shown in Table 4-4. However, most of the time users will want to allow MPC to provide the default values for project files.

Table 4-4 Component Names and Default Values

Name	Default Value
`Source_Files`	Defaults to all files in the directory that have the following extensions: `cpp`, `cxx`, `cc`, `c`, and `C`.
`Header_Files`	Defaults to all files in the directory that have the following extensions: `h`, hpp, `hxx`, and `hh`.
`Inline_Files`	Defaults to all files in the directory that have the following extensions: `i` and `inl`.
`Template_Files`	Defaults to all files in the directory that end in the following: `_T.cpp`, `_T.cxx`, `_T.cc`, `_T.c`, and `_T.C`.
`Documentation_Files`	Defaults to all files in the directory that match the following: `README`, `readme`, `.doc`, `.html` and `.txt`.
`Resource_Files`	Defaults to all files in the directory that match the project name and have an `rc` extension.

If a component is not specified in the mpc file, the default value will be used. To disallow a particular set of files that may exist in the directory, you must declare an empty set of the particular component type.

Each component name accepts two forms. The first form is a simple list of files within the construct.

```
Source_Files {
  file1.cpp
  file2.cpp
}
```

The second form is a complex list of files within named blocks.

```
Source_Files(MACRO_NAME) {
  BlockA {
    file1.cpp
    file2.cpp
  }
  BlockB {
    file3.cpp
    file4.cpp
  }
}
```

The second form allows the user to logically group the files to make future maintenance easier. Using this form has the effect of visually grouping files in the generated project file for the `em3`, `gnuace`, `vc6`, `vc7`, `vc71` and `vc8` project types.

If a file is listed in the `Source_Files` component list and a corresponding header or inline file exists in the directory, it is added to the corresponding component list unless it is already listed.

Verbatim Clause

The `verbatim` construct can be used to place text into a generated project file verbatim. The `verbatim` syntax is as follows:

```
verbatim(<project type>, <location>) {
 ..
}
```

When MPC is generating a project of type `<project type>` and encounters a `marker` in the template file (see Table 4-8 on page 67) that matches the `<location>` name, it will place the text found inside the construct directly into the generated project. If the text inside the construct requires that white space be preserved, each line must be enclosed in double quotes. The following `verbatim` example would result in `gnuace` generated projects having a rule at the bottom of the GNUmakefile where the `all:` target depends on `foo`.

```
verbatim(gnuace, bottom) {
  all: foo
}
```

Specific Clause

The `specific` keyword can be used to define assignments that are specific to a particular project type. This will allow platform or OS-specific values to be placed into a project. For example, on one platform you may want to link in a library named `qt-mt`, but on another you need to link in `qt-mt230nc`.

```
specific(bmake, nmake, vc6, vc7, vc71, vc8) {
  lit_libs += qt-mt230nc
} else {
  lit_libs += qt-mt
}
```

If an else clause is provided, it is required to be on the same line as the closing curly brace. You may also negate the project type (using '!') which will cause the specific to be evaluated for all types except the type specified.

If a keyword used within a `specific` section is not recognized as a valid MPC keyword, it is interpreted to be template value modifier. In this situation, this construct works exactly the same way as the `-value_template` command line option (see Table 4-2 on page 37).

Conditional Clause

This scope allows addition of source files conditionally based on a particular project type. The syntax is as follows:

```
conditional(<project type> [, <project type> ...]) {
  source1.cpp
  ...
}

conditional(<project type> [, <project type> ...]) {
  source1.cpp
  ...
} else {
  source2.cpp
  ...
}
```

If the else is provided, it is required to be on the same line as the closing curly brace. You may also negate the project type (using '!') which will cause the conditional to be evaluated for all types except the type specified.

Custom Types and Build Rules

MPC allows you to define your own custom file types to support a variety of custom build rules. Below is an example of a custom definition.

```
project {
  Define_Custom(MOC) {
    automatic        = 0
    command          = $(QTDIR)/bin/moc
    output_option    = -o
    inputext         = .h
    pre_extension    = _moc
    source_outputext = .cpp
    keyword mocflags = commandflags
  }

  // Custom Component
  MOC_Files {
    QtReactor.h
  }

  Source_Files {
    QtReactor_moc.cpp
  }
}
```

The above example defines a custom file type, "`MOC`", that describes basic information about how to process the input files and what output files are created. Once the custom file type is defined, `MOC_Files` can be used to specify the input files for this new file type.

Table 4-5 contains the keywords that can be used within the scope of `Define_Custom`.

Table 4-5 Define_Custom Keywords

Keyword	Description
automatic	If set to 1, then attempt to automatically determine which files belong to the set of input files for the custom type. If set to 0, then no files are automatically added to the input files. If omitted, then automatic is assumed to be 1. Custom file types that are automatic will have the side effect of possibly adding files to `Source_Files`, `Inline_Files`, `Header_Files`, `Template_Files`, `Resource_Files` and `Documentation_Files` depending on which extension types the command generates.
command	The name of the command that should be used to process the input files for the custom type.

Table 4-5 Define_Custom Keywords

Keyword	Description
commandflags	Any options that should be passed to the command.
dependent	If this is given a value, then a dependency upon that value will be placed upon all of the generated files. The default for this is unset and no dependency will be generated.
inputext	This is a comma separated list of input file extensions that belong to the command.
keyword <name>	This is a special assignment that allows the user to map <name> into the project level namespace. The value (if any) that is assigned to this construct must be one of the keywords that can be used within a Define_Custom clause. The result of this assignment is the ability modify the value of keywords that are normally only accessible within the scope of a custom component (e.g. `command`, `commandflags`, etc.).
libpath	If the command requires a library that is not in the normal library search path, this keyword can be used to ensure that the command is able to find the library that it needs to run.
output_option	If the command takes an option to specify a single file output name, then set it here. Otherwise, this should be omitted.
pch_postrule	If this is set to 1, then a rule will be added to the custom rule that will modify the source output files to include the precompiled header file.
postcommand	This allows users to create arbitrary commands that will be run after the main command is run to process the custom input files.
pre_extension	If the command produces multiple files of the same extension, this comma separated list can be used to specify them. For example, `tao_idl` creates two types of files per extension (`C.h`, `S.h`, `C.cpp`, `S.cpp`, etc.) This applies to all extension types.
source_pre_extension	This is the same as pre_extension except that it only applies to source_outputext.
inline_pre_extension	This is the same as pre_extension except that it only applies to inline_outputext.
header_pre_extension	This is the same as pre_extension except that it only applies to header_outputext.
template_pre_extension	This is the same as pre_extension except that it only applies to template_outputext.

Table 4-5 Define_Custom Keywords

Keyword	Description
resource_pre_extension	This is the same as pre_extension except that it only applies to resource_outputext.
documentation_pre_extension	This is the same as pre_extension except that it only applies to documentation_outputext.
pre_filename	The syntax for this is the same as `pre_extension`, but the values specified are prepended to the file name instead of the extension. This applies to all extension types.
source_pre_filename	This is the same as pre_filename except that it only applies to source_outputext.
inline_pre_filename	This is the same as pre_filename except that it only applies to inline_outputext.
header_pre_filename	This is the same as pre_filename except that it only applies to header_outputext.
template_pre_filename	This is the same as pre_filename except that it only applies to template_outputext.
resource_pre_filename	This is the same as pre_filename except that it only applies to resource_outputext.
documentation_pre_filename	This is the same as pre_filename except that it only applies to documentation_outputext.
source_outputext	This is a comma separated list of possible source file output extensions. If the command does not produce source files, then this can be omitted.
inline_outputext	This is a comma separated list of possible inline file output extensions. If the command does not produce inline files, then this can be omitted.
header_outputext	This is a comma separated list of possible header file output extensions. If the command does not produce header files, then this can be omitted.
template_outputext	This is a comma separated list of possible template file output extensions. If the command does not produce template files, then this can be omitted.
resource_outputext	This is a comma separated list of possible resource file output extensions. If the command does not produce resource files, then this can be omitted.
documentation_outputext	This is a comma separated list of possible documentation file output extensions. If the command does not produce documentation files, then this can be omitted.

Table 4-5 Define_Custom Keywords

Keyword	Description
generic_outputext	If the command does not generate any of the other output types listed above, then the extensions should be listed under this.

There is a special interaction between custom components and the source, header and inline components. If a `custom` definition is set to be "automatic" and custom component files are present but not specified, the default custom generated names are added to the source, header and inline component lists unless those names are already listed (or partially listed) in those component lists. See "Custom Types and Build Rules" on page 52 for more information about defining your own custom type.

Particular output extensions for custom build types are not required. However, at least one output extension type is required for MPC to generate a target. Your command does not necessarily have to generate output, but an extension type is required if you want the input file to be processed during the project compilation.

If the custom output can not be represented with the above output extension keywords (*_outputext) and you have knowledge of the output files *a priori*, you can represent them with the '>>' construct.

Below is an example that demonstrates the use of '>>'. The command takes an input file name of foo.prp and produces two files that have completely unrelated filenames, `hello.h` and `hello.cpp`.

```
project {
  Define_Custom(Quogen) {
    automatic           = 0
    command             = perl quogen.pl
    commandflags        = --debuglevel=1 --language=c++ \
                          --kernel_language=c++
    inputext            = .prp
    keyword quogenflags = commandflags
  }

  Quogen_Files {
    foo.prp >> hello.h hello.cpp
  }

  Source_Files {
    hello.cpp
  }
}
```

You can use the '<<' construct to represent dependencies for specific custom input file. For instance, in the above example, assume that foo.prp depends upon foo.in, we would represent this by adding << foo.in as shown below.

```
Quogen_Files {
  foo.prp >> hello.h hello.cpp << foo.in
}
```

An additional construct can be used within the scope of a `Define_Custom`. This construct is called `optional`, and can be used to represent optional custom command output that is dependent upon particular command line parameters passed to the custom command.

```
project {
  Define_Custom(TEST) {
    optional(keyword) {
      flag_keyword(option) += value [, value]
    }
  }
}
```

In the above fragment, `keyword` can be any of the `pre_extension`, `pre_filename` keywords or any of the keywords that end in `_outputext`. The `flag_keyword` can be any of the custom definition keywords, however only `commandflags` has any functional value. The `flag_keyword` value is searched for the `option` value contained inside the parenthesis. If it is found the `value` or `values` after the += are added to the list specified by `keyword`. This can also be negated by prefixing the `option` with an exclamation point (!).

The example below shows how the `optional` construct is used by the custom definition for the `tao_idl` command (see `ACE_wrappers/bin/MakeProjectCreator/config/taoidldefaults.mpb`). The `-GA` option causes `tao_idl` to generate an additional source file (based on the idl file name) with an `A.cpp` extension. The `-Sc` option causes `tao_idl` to suppress the generation of `S_T` related files.

```
Define_Custom(IDL) {
  ...
  inputext             = .idl
  source_pre_extension = C, S
  header_pre_extension = C, S
  inline_pre_extension = C, S
  source_outputext     = .cpp, .cxx, .cc, .C
  header_outputext     = .h, .hpp, .hxx, .hh
```

```
    inline_outputext     = .inl, .i
    keyword idlflags     = commandflags

    optional(source_pre_extension) {
      commandflags(-GA) += A
    }
    optional(template_outputext) {
      commandflags(!-Sc) += S_T.cpp, S_T.cxx, S_T.cc, S_T.C
    }
    optional(header_pre_extension) {
      commandflags(!-Sc) += S_T
    }
    optional(inline_pre_extension) {
      commandflags(!-Sc) += S_T
    }
  }
```

For custom file types, there are a few keywords that can be used within the custom file type component lists: `command`, `commandflags`, `dependent`, `gendir`, `postcommand`, and `recurse`.

The `recurse` keyword works as described in Table 4-3, “Assignment Keywords”.

The `command`, `commandflags`, `dependent` and `postcommand` keywords can be used to augment or override the value defined in the `Define_Custom` section.

The `gendir` keyword can be used (only if `output_option` is set in `Define_Custom`) to specify the directory in which the generated output will go. Here is an example:

```
MOC_Files {
  commandflags += -nw
  gendir = moc_generated
  QtReactor.h
}

Source_Files {
  moc_generated/QtReactor_moc.cpp
}
```

In the above example, the `-nw` option is added to `commandflags` and the generated file (`QtReactor_moc.cpp`) is placed in the `moc_generated` directory. If the `MOC` custom definition did not have an `output_option` setting, then options would need to be added to `commandflags` or a

postcommand would need to be defined to ensure that the output actually went into the moc_generated directory.

Custom Post Command

When defining a postcommand as part of a Define_Custom, a few pseudo template variables are available to provide some flexibility. The following table shows the pseudo template variables that can be accessed only from the postcommand. Please note that <% and %> are part of the syntax.

Table 4-6 Post Command Pseudo Variables

Variable	Description
<%input%>	The input file for the original command.
<%input_basename%>	The basename of the input file for the original command.
<%input_noext%>	The input file for the original command with the extension stripped off.
<%input_ext%>	This gives the file extension of the input file (if there is one).
<%output%>	The output file created by the original command.
<%output_basename%>	The basename of the output file for the original command.
<%output_noext%>	The output file created by the original command with the extension stripped off.
<%output_ext%>	This gives the file extension of the output file (if there is one).
The output file can be referenced as a generic output file, or it can be referenced as a component file using one of the following variables. If it does not match the particular type the value will be empty.	
<%source_file%>	The output file if it has a source file extension.
<%template_file%>	The output file if it is a template file.
<%header_file%>	The output file if it has a header file extension.
<%inline_file%>	The output file if it has an inline file extension.
<%documentation_file%>	The output file if it is a documentation file.
<%resource_file%>	The output file if it has a resource file extension.

The following table describes the pseudo template variables that can be used in the `command`, `commandflags`, `dependent`, `output_option` and `postcommand` settings.

Table 4-7 Common Pseudo Variables

Variable	Description
`<%temporary%>`	A temporary file name. The generated temporary file name contains no directory portion and is the same for each use within the same variable setting.
`<%cat%>`	A platform non-specific command to print a file to the terminal.
`<%cp%>`	A platform non-specific command to copy a file.
`<%mkdir%>`	A platform non-specific command to make a directory.
`<%mv%>`	A platform non-specific command to move a file.
`<%rm%>`	A platform non-specific command to delete a file.
`<%nul%>`	A platform non-specific null device.
`<%gt%>`	A platform and project non-specific representation of a greater than sign.
`<%lt%>`	A platform and project non-specific representation of a less than sign.
`<%quote%>`	A project non-specific representation of a double quote.
`<%and%>`	A platform and project non-specific representation of a command conditional and.
`<%or%>`	A platform and project non-specific representation of a command conditional or.

4.3.2.3 The Feature File

The term feature, as used by MPC, describes different concepts or external software that a project may require in order to build properly. The feature file determines which features are enabled or disabled which has a direct effect on whether or not MPC generates a project.

It supports the standard comment (`//`) and assignment of numbers to feature names. These feature names will correspond to values given to the `requires` and `avoids` keywords in mpc files.

If a feature is not listed in the feature file or is listed with a boolean value of true (1), that feature is enabled. If a feature is listed and has a boolean value of false (0), that feature is disabled.

If a feature name is listed in the `requires` value for a particular project and that feature is enabled, that project will be generated. If the feature is not enabled, the project will not be generated.

The opposite holds true for the `avoids` keyword. If a feature name is listed in the `avoids` value for a project and the feature is disabled, that project will be generated. If the feature is enabled, the project will not be generated.

The global feature file for MPC contains the following values.

```
boost = 0
mfc = 0
qt = 0
rpc = 0
zlib = 0
zzip = 0
```

In the above contents, `boost`, `mfc`, `qt`, `rpc`, `zlib` and `zzip` are disabled for each project generated. If these values do not suit your needs, then you must do one of three things:

- Create a project specific feature file in the `config` directory (ex. `make.features`) to set features for a particular project type.
- Create a `default.features` file in the `config` directory that contains the feature set you need.
- Create a feature file anywhere you like with the features you want and use the `-feature_file` option to specify the location.
- Use the -features option to dynamically modify the feature settings.

Generated projects will have a combination of features specified in the `global.features` file as well as in your feature file. Therefore, if a feature is disabled in the global file and you want to enable it, you must explicitly enable it in your feature file.

4.3.2.4 Feature Projects

A feature project contains information as a project would, but can only be a base project and will only be added to a sub project if the features that it requires are enabled or the features that it avoids are disabled.

A feature definition requires at least one feature name. A name by itself specifies that the feature must be enabled. A '!' in front of the feature name

indicates that the feature must be disabled. There may be more than one comma separated feature listed between the parenthesis.

The following example show how to declare a feature project.

```
// ziparchive.mpb
feature(ziparchive) {
  includes += $(ZIPARCHIVEROOT)
  libpaths += $(ZIPARCHIVEROOT)/lib
  libs     += ziparch
}
```

With this example, any project that inherits from the `ziparchive` base feature project will contain the project information only if the `ziparchive` feature is enabled.

4.3.3 Defaults

MPC has been designed to minimize the amount of maintenance that goes into keeping build tool files up-to-date with the project. If your source code is organized *properly*, the maintenance of your mpc files should be minimal.

With the use of inheritance and proper code arrangement, an mpc file for a TAO related project may be as simple as:

```
project: taoserver {
}
```

This project definition could be used to generate a project for a TAO server with multiple idl, header and source files.

The idea of *proper* source layout is basically summarized as *one directory per binary target.* If only the files that pertain to a single target are located in the directory with the mpc file, then the MPC defaults will satisfy most project needs.

Of course, it will not always be possible or desirable to organize your project code in this fashion, so all defaulting behavior can be overridden. The next sections describe the default behaviors of MPC and how to override them.

4.3.3.1 Source Files

New source files are added and others are removed quite often in a developing project. If the `Source_Files` component is left out of an mpc file, then MPC will assume that any file matching one of the *source* extensions is to be included in the project. For most project types, the source extensions are:

.cpp, .cxx, .cc, .c and .C. Only the following extensions are considered source extensions: .cpp, .cxx and .c for the vc6 project type as Visual C++ 6.0 does not understand files with the .cc or .C extension.

4.3.3.2 Template Files

MPC assumes that any file matching one of the *template* extensions is to be included in the project if the Template_Files component is left out of an mpc file. For most project types, the template extensions are: _T.cpp, _T.cxx, _T.cc, _T.c and _T.C. However, only the _T.cpp and _T.cxx extensions are considered template extensions for the vc6 project type.

If the Source_Files component is defaulted, and a file is explicitly listed in the Template_Files section that happens to appear to MPC as a source file (i.e. has a source file extension, but does not have _T directly before it), MPC will automatically exclude it from the Source_Files component.

4.3.3.3 Inline Files

As with source files, the Inline_Files component can be left out of an mpc file to allow it to generate defaults. Files that match the .i and .inl extensions are considered inline files.

The Inline_Files component has a special interaction with the Source_Files component. If the Source_Files component has files listed and the Inline_Files component is omitted, then each source file is *matched* to an inline file. If the matching inline file is found or would be generated from a custom command, it is added to the Inline_Files component list.

4.3.3.4 Header Files

As with source files, the Header_Files component can be left out of an mpc file to allow it to generate defaults. Files that match the .h, .hpp, .hxx, and .hh extensions are considered header files.

The Header_Files component has a special interaction with the Source_Files component. If the Source_Files component has files listed and the Header_Files component is omitted, then each source file is *matched* to a header file. If the matching header file is found or would be generated from a custom command, then it is added to the Header_Files component list.

4.3.3.5 Documentation Files

The `Documentation_Files` component, if omitted, will default to all files that end in the following: `README`, `readme`, `.doc`, `.html` and `.txt`.

4.3.3.6 Resource Files

The `Resource_Files` component, if omitted, will default to only the files that end in `.rc` and are similar to the name of the project. For example, if a directory contains three `.rc` files and the project name is `foo`, only the `.rc` files that contain the word `foo` will automatically be added to the `Resource_Files` component list.

4.3.3.7 Custom Defined Files

The `Custom Defined Files` components have a special interaction with the `Source_Files` component. If the custom command generates source files and has the `automatic` setting set to 1, they will automatically be added to the `Source_Files` component list. If any of the files listed in the `Source_Files` components list match any of the generated source file names, then none of the generated source file names will be automatically added to the `Source_Files` components list.

4.3.3.8 Example MPC File

The example below uses the directory contents of `$TAO_ROOT/orbsvcs/performance-tests/RTEvent/lib` to illustrate the simplicity of mpc files:

```
Auto_Disconnect.cpp       Loopback_Supplier.h     RTEC_Initializer.cpp
Auto_Disconnect.h         Low_Priority_Setup.cpp  RTEC_Initializer.h
Auto_Disconnect.inl       Low_Priority_Setup.h    rtec_perf_export.h
Auto_Functor.cpp          Low_Priority_Setup.inl  RTEC_Perf.mpc
Auto_Functor.h            Makefile                RTPOA_Setup.cpp
Auto_Functor.inl          ORB_Holder.cpp          RTPOA_Setup.h
Client_Group.cpp          ORB_Holder.h            RTPOA_Setup.inl
Client_Group.h            ORB_Holder.inl          RTServer_Setup.cpp
Client_Group.inl          ORB_Shutdown.cpp        RTServer_Setup.h
Client_Options.cpp        ORB_Shutdown.h          RTServer_Setup.inl
Client_Options.h          ORB_Shutdown.inl        Send_Task.cpp
Client_Pair.cpp           ORB_Task_Activator.cpp  Send_Task.h
Client_Pair.h             ORB_Task_Activator.h    Send_Task_Stopper.cpp
Client_Pair.inl           ORB_Task_Activator.inl  Send_Task_Stopper.h
Consumer.cpp              ORB_Task.cpp            Send_Task_Stopper.inl
Consumer.h                ORB_Task.h              Servant_var.cpp
```

```
Control.cpp               ORB_Task.inl            Servant_var.h
Control.h                 Peer_Base.cpp           Servant_var.inl
EC_Destroyer.cpp          Peer_Base.h             Shutdown.cpp
EC_Destroyer.h            PriorityBand_Setup.cpp  Shutdown.h
EC_Destroyer.inl          PriorityBand_Setup.h    Shutdown.inl
Federated_Test.idl        PriorityBand_Setup.inl  Supplier.cpp
Implicit_Deactivator.cpp  RIR_Narrow.cpp          Supplier.h
Implicit_Deactivator.h    RIR_Narrow.h            SyncScope_Setup.cpp
Implicit_Deactivator.inl  RT_Class.cpp            SyncScope_Setup.h
Loopback_Consumer.cpp     RT_Class.h              SyncScope_Setup.inl
Loopback_Consumer.h       RT_Class.inl            TAO_RTEC_Perf.dsp
Loopback.cpp              RTClient_Setup.cpp      TAO_RTEC_Perf.dsw
Loopback.h                RTClient_Setup.h        Task_Activator.cpp
Loopback_Pair.cpp         RTClient_Setup.inl      Task_Activator.h
Loopback_Pair.h           RTCORBA_Setup.cpp       Task_Activator.inl
Loopback_Pair.inl         RTCORBA_Setup.h
Loopback_Supplier.cpp     RTCORBA_Setup.inl
```

The following mpc file (`RTEC_Perf.mpc`) shows the simple and small number of lines required to generate usable build tool project files.

```
project(RTEC_Perf): strategies, rtcorbaevent, minimum_corba {
  sharedname = TAO_RTEC_Perf
  idlflags  += -Wb,export_macro=TAO_RTEC_Perf_Export \
               -Wb,export_include=rtec_perf_export.h
  dllflags  += TAO_RTEC_PERF_BUILD_DLL

  Template_Files {
    Auto_Disconnect.cpp
    Auto_Functor.cpp
    Low_Priority_Setup.cpp
    RIR_Narrow.cpp
    Servant_var.cpp
    Shutdown.cpp
    Task_Activator.cpp
  }
}
```

A line-by-line explanation of the example mpc file is listed below.

```
project(RTEC_Perf): strategies, rtcorbaevent, minimum_corba {
```

The first line declares a project named `RTEC_Perf` that inherits from the base projects listed after the colon.

```
  sharedname = TAO_RTEC_Perf
```

Line 2 determines that the project is a library and the library name is `TAO_RTEC_Perf`.

```
idlflags  += -Wb,export_macro=TAO_RTEC_Perf_Export \
             -Wb,export_include=rtec_perf_export.h
```

Lines 3-4 add to the flags passed to the IDL compiler when processing the idl files.

```
dllflags  += TAO_RTEC_PERF_BUILD_DLL
```

The next line adds `TAO_RTEC_PERF_BUILD_DLL` to the `dllflags`, which defines a macro that is used by the `rtec_perf_export.h` header file.

```
Template_Files {
  Auto_Disconnect.cpp
  Auto_Functor.cpp
  Low_Priority_Setup.cpp
  RIR_Narrow.cpp
  Servant_var.cpp
  Shutdown.cpp
  Task_Activator.cpp
}
```

Lines 7-15 name the listed cpp files as part of the `Template_Files`.

You may have noticed that there isn't much to the file above. With the default behaviors that are built into MPC, there does not need to be. We rely on the defaults to determine the values of `IDL_Files`, `Source_Files`, `Inline_Files`, and `Header_Files`. Since the template files do not match the MPC built-in defaults, we must explicitly list them. We also rely on inheritance to get many of the TAO-related options.

4.4 Adding a New Type

If MPC does not support a particular build tool, you may want to consider adding a new project type. For instance, support could be added to MPC for Boost Jam, Eclipse, Xcode and many others. To do so will require knowledge of the MPC input files, as well as Object Oriented Perl.

4.4.1 Input File Syntax

This section describes the syntax of the files that are used during project generation.

4.4.1.1 Template Files (mpd)

Template files make up the bulk of what MPC puts into each generated project file. They provide the plain text and the layout of the data provided by the mpc files, using various template directives.

Template directives are declared using a <% %> construct. This construct is used to create if statements, for loops and to access variables. One thing to note is that any text, including white space, that is not enclosed within <% %> is left untouched and is passed directly into the generated project file.

An `if` statement can appear on a single line or it can span multiple lines. For example, the following line:

```
<%if(exename)%>BIN = <%exename%><%else%>LIB = <%sharedname%><%endif%>
```

is equivalent to:

```
<%if(exename)%>
BIN = <%exename%>
<%else%>
LIB = <%sharedname%>
<%endif%>
```

A `foreach` statement can also appear on a single line or can span multiple lines. As described below in the keywords section, the `foreach` statement evaluates the variable in a space-separated list context.

There are a couple of ways to write a `foreach` loop. The first and preferred way is to name the loop variable and then list each variable to be evaluated.

```
FILES=<%foreach(fvar, idl_files source_files header_files)%> <%fvar%><%endfor%>
```

The second way is to let the `foreach` statement determine the loop variable. With this style, each value can be accessed via the first variable name passed to the `foreach` with the trailing '`s`' removed.

```
FILES=<%foreach(idl_files source_files header_files)%> <%idl_file%><%endfor%>
```

Note that the <%idl_file%> variable will contain each individual value of the idl_files, source_files and header_files list. If the variable in the foreach does not end in 's', the variable of the same name within the foreach will contain each individual value, e.g.,

```
<%foreach(filelist)%> <%filelist%><%endfor%>
```

Table 4-8 lists keywords that can appear in template files.

Table 4-8 Template File Keywords

Keyword	Description
if	Used to determine if a variable is defined. The not operator (!) can be used to invert the if check. This construct will only check for values defined within an mpc or mpt file. Default values (even those implemented by the project creators) are not considered in the if statement.
else	Used with the if statement. An else block will be evaluated if the statement does not evaluate to true.
endif	Used with the if statement. This ends an if or if/else block.
noextension	Evaluates the variable name value as a file name and removes the extension from that value including the period.
dirname	Evaluates the variable name and removes the basename from that value.
basename	Evaluates the variable name and removes the directory portion from that value.
basenoextension	This is similar to basename except that the extension is also removed from the variable name value.
foreach	The given variable names are evaluated in a list context which is space separated.
forfirst	Used with foreach. The literal value passed to forfirst will be placed on the first iteration of foreach.
fornotfirst	Used with foreach. The literal value passed to fornotfirst will be placed on each iteration of foreach except for the first.
forlast	Used with foreach. The literal value passed to forlast will be placed on the last iteration of foreach.

Table 4-8 Template File Keywords

Keyword	Description
`fornotlast`	Used with `foreach`. The literal value passed to `fornotlast` will be placed on each iteration of `foreach` except for the last.
`endfor`	Used with `foreach`. This ends `foreach` block.
`comment`	The value passed to comment is ignored and can be any set of characters, except a new line or a closing parenthesis.
`flag_overrides`	This is directly related to overriding the project-wide settings in an mpc file. It takes two variable names that are comma separated. The first corresponds to a file name and the second is any variable name.
`marker`	This is directly related to the `verbatim` keyword from the mpc syntax. This can be used to designate markers within a template. Ex. `<%marker(local)%>`.
`uc`	Return the given variable value in all upper case characters.
`lc`	Return the given variable value in all lower case characters.
`ucw`	Return the given variable value with the first letter of each word in upper case. Words are separated by spaces or underscores.
`normalize`	Convert dashes, slashes, dollar signs, parenthesis and dots in the given variable value to underscores.
`reverse`	This function reverses the order of the array parameter values.
`sort`	This function sorts the array parameter values.
`uniq`	This function returns the unique set of the array parameter values.
`multiple`	This function returns true if the array parameter contains multiple values.
`starts_with`	This function returns true if the variable value (first parameter) starts with the regular expression (second parameter).
`ends_with`	This function returns true if the variable value (first parameter) ends with the regular expression (second parameter).

Table 4-8 Template File Keywords

Keyword	Description
`contains`	This function returns true if the variable value (first parameter) contains the regular expression (second parameter).

Table 4-9 lists special names that can be used as variables in some template files. The variables listed in Table on page 59 can be used as well (except for <%temporary%>).

Table 4-9 Special Values used in Template Files

Value	Description
`custom_types`	Contains a list of the custom build types. See "Custom Types" on page 70 for more details.
`cwd`	The full current working directory.
`forcount`	This only has a value within the context of a `foreach` and provides a 1 based count of the index of the elements in `foreach`.
`project_name`	This variable contains the name of the current project being generated.
`project_file`	This variable contains the name of the output file for the current project being generated.
`ciao`	Implemented by the GNUACE project creator module, specifies that the project uses CIAO.
`cppdir`	This value is implemented by the BMake project creator modules. It returns a semicolon separated list of directories taken from each value in the `Source_Files` list.
`rcdir`	This value is implemented by the BMake project creator modules. It returns a semicolon separated list of directories taken from each value in the `Resource_Files` list.
`make_file_name`	This value is implemented by the VC6 and EM3 project creator modules. It returns the project name with the make file extension that corresponds to the particular project type.
`tao`	Implemented by the GNUACE project creator module, specifies that the project uses TAO.
`guid`	This value is implemented by the VC7 project creator module. It returns a `guid` value based on the project that is usable within VC7 project files.

Table 4-9 Special Values used in Template Files

Value	Description
`vcversion`	This value is implemented by the VC7ProjectCreator. It returns the version number of the type of project being generated. 7.00 is return for vc7, 7.10 is return for vc71 and 8.00 is returned for vc8.
`vpath`	This value is implemented by the GNUACEProjectCreator. It returns a value, based on the location of the source files, that specifies the `VPATH` setting for GNU Make.

Custom Types

To support multiple custom build types, a special keyword was introduced. The `custom_types` keyword is used to access the list of custom types defined by the user. In a `foreach` context, each custom type can be accessed through the `custom_type` keyword.

A variety of information is available from each `custom_type` through the `->` operator. The input files, input extensions, command, command output option, command flags, and output file directory are all accessible through the field names that correspond to the particular type.

The input files associated with the custom type are accessed through `custom_type->input_files`. Each input file has a set of output files associated with it which can be accessed in a `foreach` context through `custom_type->input_file->output_files`. The custom type fields are listed in Table 4-10.

Table 4-10 Custom Type Fields

Value	Description
`command`	The command used for the custom type.
`commandflags`	The command options not including the output option.
`dependent`	This setting determines the command upon which custom generated files should depend.
`gendir`	The output directory associated with a particular input file. This field has no meaning when accessed directly through the `custom_type`. It should always be used within the context of a `flag_overrides` (See Table 4-8).
`input_files`	The input files associated with the custom type.
`inputexts`	The input file extensions associated with the custom type.
`libpath`	The library path setting for the command.

Table 4-10 Custom Type Fields

Value	Description
`output_option`	The optional command output option.
`pch_postrule`	This setting determines whether the command needs assistance in supporting precompiled headers.
`postcommand`	Allows a user to execute arbitrary commands after the main command is run to generate the output file.

The example below, which creates generic makefile rules for building custom input files, shows basic use of the custom type and the various fields that can be accessed. The main limitation with the `custom_types` keyword, as can be seen below, is that the `foreach` variable cannot be named as stated on page 66.

```
<%if(custom_types)%>
<%foreach(custom_types)%>
<%foreach(custom_type->input_files)%>
<%foreach(custom_type->input_file->output_files)%>
<%custom_type->input_file->output_file%>: <%custom_type->input_file%>
   <%custom_type->command%> <%custom_type->commandflags%> $@

<%endfor%>
<%endfor%>
<%endfor%>
<%endif%>
```

Grouped Files

File grouping is part of the syntax of mpc files. If a set of files are grouped within the mpc file, they can be accessed as a group within the mpd file.

Files (such as `Source_Files`, `Header_Files`) can be grouped together as shown on page 50. Within the mpd file, the different components can be accessed by prepending `grouped_` to the component (`grouped_source_files`, `grouped_header_files`, etc.)

Table 4-11 Grouped Files Field Names

Field Name	Description
`files`	The input files associated with the group.
`component_name`	The name of the set of multiple groups of files.

The example below, which creates make macros for each file group, shows basic use of grouping and the fields that can be accessed. The main limitation with file grouping, as can be seen below, is that the `foreach` variable cannot be named as stated on page 66. The following example involves source files, but any of the components listed in 4.3.2.2 can be used.

```
<%if(grouped_source_files)%>
<%comment(Get back each set of grouped files)%>
<%foreach(grouped_source_files)%>
<%comment(This will provide the name of the group)%>
<%grouped_source_file%> = \
<%comment(Get all the source files in a single group)%>
<%foreach(grouped_source_file->files)%>
  <%grouped_source_file->file)%><%fornotlast(" \\")%>
<%endfor%>
<%endfor%>

ifndef <%grouped_source_files->component_name%>
  <%grouped_source_files->component_name%> = \
<%foreach(grouped_source_files)%>
    <%grouped_source_file%><%fornotlast(" \\")%>
<%endfor%>
endif
<%endif%>
```

4.4.1.2 Template Input Files (mpt)

Template input files provide build tool specific information that is common to all projects, such as compiler switches, intermediate directories, compiler macros, etc. Each project type can provide template input files for dynamic libraries, static libraries, dynamic executables and static executables. However, none of these are actually required by MPC.

The template input files are more free-form than the other MPC file types. It is similar to the mpc syntax except that there is no project definition and there is only one keyword. The keyword, `conditional_include`, is used to include other mpt files if they can be found in the MPC include search path. If the name listed in double quotes after `conditional_include` is not found, it is ignored and no warning is produced. The `mpt` extension is automatically added to the name provided.

The template input files contain variable assignments and collections of variable assignments. A variable assignment is of the form:

```
variable_name = value1 "value 2"
variable_name += another_value
```

This variable can then be used within the corresponding mpd file.

Variable assignments can be grouped together and named within the mpt file and used as scoped variables within the mpd file. The following example shows the use of collections of variable assignments.

```
// mpt file
configurations = Release Debug
common_defines = WIN32 _CONSOLE

Release {
  compile_flags = /W3 /GX /O2 /MD /GR
  defines = NDEBUG
}

Debug {
  compile_flags = /W3 /Gm /GX /Zi /Od /MDd /GR /Gy
  defines = _DEBUG
}

conditional_include "vcfullmacros"
```

Below is the portion of the mpd file that would use the information provided in the mpt file above.

```
<%foreach(configurations)%>
Name = <%configuration%>
<%compile_flags%><%foreach(defines common_defines)%> /D <%define%>=1<%endfor%>

<%endfor%>
```

The following output is generated from the above example:

```
Name = Release
/W3 /GX /O2 /MD /GR /D NDEBUG=1 /D WIN32=1 /D _CONSOLE=1

Name = Debug
/W3 /Gm /GX /Zi /Od /MDd /GR /Gy /D _DEBUG=1 /D WIN32=1 /D _CONSOLE=1
```

If a `foreach` variable value corresponds to a variable group name, that variable group is available within the scope of that `foreach`.

4.4.2 A Simple Example

We will discuss what it would take to add support for a fictional build tool throughout this section. The diagram on page 35 shows the relationship between the template and project creator discussed below.

4.4.2.1 Template

The best thing to do is to start with the template. The template is the most important piece when adding a new project type. It basically tells MPC how to lay out all of the information it gathers while processing an mpc file. The template file will have a mixture of plain text and the mpd syntax described in 4.4.1.1. Here is our sample `fictional.mpd`.

```
//=============================================================================
// This project has been generated by MPC.
// CAUTION! Hand edit only if you know what you are doing!
//=============================================================================

// Section 1 - PROJECT OPTIONS
ctags:*
debugSwitches:-nw
//end-proj-opts

// Section 2 - MAKEFILE
Makefile.<%project_name%>

// Section 3 - OPTIONS
//end-options

// Section 4  - TARGET FILE
<%if(exename)%>
<%exename%>
<%else%>
<%if(sharedname)%>
<%sharedname%>
<%else%>
<%if(staticname)%>
<%staticname%>
<%endif%>
<%endif%>
<%endif%>

// Section 5  - SOURCE FILES
<%foreach(source_files)%>
<%source_file%>
<%endfor%>
//end-srcfiles
```

```
// Section 6  - INCLUDE DIRECTORIES
<%foreach(includes)%>
<%include%>
<%endfor%>
//end-include-dirs

// Section 7 - LIBRARY DIRECTORIES
<%foreach(libpaths)%>
<%libpath%>
<%endfor%>
//end-library-dirs

// Section 8  - DEFINITIONS
<%foreach(macros defines)%>
-D<%macro%>
<%endfor%>
<%if(pch_header)%>
<%foreach(pch_defines)%>
-D<%pch_define%>
<%endfor%>
<%endif%>
//end-defs-pool

// Section 9  - C FLAGS
<%cflags("-g")%>

// Section 10  - LIBRARY FLAGS
<%libflags%>

// Section 11  - SRC DIRECTORY
.

// Section 12  - OBJ DIRECTORY
<%objdir(".")%>

// Section 13 - BIN DIRECTORY
<%if(install)%><%install%><%else%>.<%endif%>

// User targets section. Following lines will be
// inserted into Makefile right after the generated cleanall target.
// The Project File editor does not edit these lines - edit the .vpj
// directly. You should know what you are doing.
// Section 14 - USER TARGETS
<%marker(top)%>
<%marker(macros)%>
<%marker(local)%>
<%marker(bottom)%>
//end-user-targets

// Section 15 - LIBRARY FILES
```

```
<%foreach(libs lit_libs pure_libs)%>
<%lib%>
<%endfor%>
//end-library-files
```

Note that output is generated differently depending upon whether <%exename%>, <%sharedname%> or <%staticname%> is defined due to the if statements that were used with relation these variable names. Also, certain portions of the project file are only generated if particular variables are set.

4.4.2.2 Project Creator

Next, you would write the `FictionalProjectCreator.pm`. It may be best to start with a copy of the `MakeProjectCreator.pm` and edit it. Change the package name to `FictionalProjectCreator` and have it inherit from `MakeProjectBase` and `ProjectCreator`. Then, override the methods that are needed for this particular type.

```
package FictionalProjectCreator;

# ************************************************************
# Description   : A Fictional Project Creator
# Author        : Chad Elliott
# Create Date   : 10/01/2004
# ************************************************************

# ************************************************************
# Pragmas
# ************************************************************

use strict;

use MakeProjectBase;
use ProjectCreator;

use vars qw(@ISA);
@ISA = qw(MakeProjectBase ProjectCreator);

# ************************************************************
# Subroutine Section
# ************************************************************

sub convert_slashes {
  #my($self) = shift;
  return 0;
}
```

```
sub project_file_extension {
  #my($self) = shift;
  return '.fic';
}


sub get_dll_exe_template_input_file {
  #my($self) = shift;
  return 'fictionalexe';
}


sub get_dll_template_input_file {
  #my($self) = shift;
  return 'fictionaldll';
}


sub get_template {
  #my($self) = shift;
  return 'fictional';
}

1;
```

In our example, we inherit from the MakeProjectBase which provides some methods that are common to all "make" based project creators.

We override the `convert_slashes` method to return 0. A zero return value tells MPC not to convert slashes to back slashes (converting slashes is useful for Windows related build tools).

We then override the `project_file_extension` method to return the project file extension which is used by a method defined in the MakeProjectBase module.

Next, we override the `get_dll_exe_template_input_file` and `get_dll_template_input_file` methods. Those methods return the specific template input file names for a dynamic executable and dynamic library, respectively.

Lastly, we override the `get_template` method to return the template file name for our new project type. In our case, the method returns `fictional` which corresponds to the name of the template file we created earlier.

There are many other methods that can be overridden to change the way MPC generates output. For a complete list, see the "Virtual Methods To Be Overridden" section of the `Creator.pm` and `ProjectCreator.pm`.

4.4.2.3 Workspace Creator

The next part that you would need to write is the `FictionalWorkspaceCreator.pm`. This module is usually more code-intensive than its Project Creator counterpart.

```
package FictionalWorkspaceCreator;

# ************************************************************
# Description   : A Fictional Workspace Creator
# Author        : Chad Elliott
# Create Date   : 10/01/2004
# ************************************************************

# ************************************************************
# Pragmas
# ************************************************************

use strict;

use FictionalProjectCreator;
use WorkspaceCreator;

use vars qw(@ISA);
@ISA = qw(WorkspaceCreator);

# ************************************************************
# Subroutine Section
# ************************************************************

sub workspace_file_name {
  my($self) = shift;
  return $self->get_modified_workspace_name($self->get_workspace_name(),
                                            '.fws');
}


sub pre_workspace {
  my($self) = shift;
  my($fh)   = shift;
  my($crlf) = $self->crlf();

  print $fh '<?xml version="1.0" encoding="UTF-8"?>', $crlf,
            '<!-- MPC Command -->', $crlf,
            "<!-- $0 @ARGV -->", $crlf;
```

```
}


sub write_comps {
  my($self)     = shift;
  my($fh)       = shift;
  my($projects) = $self->get_projects();
  my(@list)     = $self->sort_dependencies($projects);
  my($crlf)     = $self->crlf();

  print $fh '<projects>', $crlf;
  foreach my $project (@list) {
    print $fh "  <project path=\"$project\"/>$crlf";
  }
  print $fh "</projects>$crlf";
}


1;
```

The first method we override from WorkspaceCreator.pm is the `workspace_file_name` method. It is used to determine the output file for the generated workspace.

Second, we override the `pre_workspace` method, which we use to print out the generic unchanging section of our generated workspace.

Lastly, we override the `write_comps` method. This method is where the bulk of the work is done in our workspace creator. A workspace creator has many sets of data available. A reference to the list of project file names can be obtained through the `get_projects` method; project-specific information can be obtained through the `get_project_info` method which returns an array reference where each array element is an array containing the project name, project dependencies and a project guid (if applicable).

4.4.2.4 MPC.pm and MWC.pm

Perhaps the easiest part of adding a new workspace and project type is modifying the `MPC.pm` and `MWC.pm` modules. Each file contains a "Subroutine Section" with a method named "`new`". This method contains an array of strings named `creators`. The `creators` array contains the names of the available project creators (`MPC.pm`) and workspace creators (`MWC.pm`). The only thing that needs to be done for our new type is to add `FictionalProjectCreator` to the `creators` list in `MPC.pm` and

`FictionalWorkspaceCreator` to the `creators` list in `MWC.pm`. It is important that the new type is not added to the beginning of the array. The first type in the array determines the default type and should not be changed. The options will automatically be updated to reflect the new type.