Xcpp Build System

David Barrett-Lennard
Cedanet Pty Ltd
Perth, Western Australia
david.barrettlennard@cedanet.com.au

March 24, 2010

Abstract

This article describes the Xcpp Build System which allows developers to conveniently and flexibly define Microsoft Visual C++ projects and solutions using simple text files written in a custom language.

1 Overview

The *Xcpp Build System* provides a convenient and flexible means to create and edit C++ projects and solutions that target the Microsoft Visual C++ Integrated Development Environment (MSVC).

In earlier versions of MSVC, projects are recorded in text files with a `dsp` extension, and workspaces in text files with a `dsw` extension. Later versions of MSVC (from Visual C++ 7.0, released in 2002) use XML files (with extension `vcproj` for project files and `sln` for solution files). A solution is essentially a workspace.

Developers don't normally edit these `.vcproj` and `.sln` XML files directly. Doing so would be inconvenient – because the format is verbose and repetitive. For example, there is no means to apply compiler switches to all configurations at once.

Instead MSVC provides wizards to create new projects and solutions, and sophisticated GUIs to edit the settings. One unfortunate result of this is that most developers don't tend to factor out the build settings across projects even though MSVC supports a concept of inheritance of project settings. It seems that normal practice tends to be dictated by the wizards, and the wizards don't promote sharing of settings amongst projects.

The Xcpp build system allows developers to create textual representations of projects and workspaces, in files with extensions `xcpj` and `xcws` respectively. These textual representations are written in a custom designed language that is simple, elegant, flexible and very concise. It promotes sharing of build settings with the result that a given project typically only needs to specify the files in that project (because all the compiler switches, linker switches and configurations are inherited using an `@import` directive).

It is claimed that the textual representation ends up being easier and simpler than the MSVC approach based on GUI editing of settings. This advantage is greatest when there are large teams and complex configuration requirements.

The Xcpp Build System *calculates* `.vcproj` and `.sln` files from the `.xcpj` and `.xcws` files. Therefore developers can edit, build and debug projects/solutions in MSVC in the normal way with no limitations whatsoever.

2 Guiding principles

2.1 Exporting targets

Typically when a DLL is formally packaged and released only the following files are *exported*:

- The public header files
- Documentation of the API, perhaps with demonstration programs.
- The `.lib` file, which is needed when static linking to the library
- The `.dll` file.

A potential disadvantage to a client is that they cannot easily debug into the DLL without the original source code. Furthermore the client has in effect only been granted read access to the DLL, because without the source code they cannot make changes to the source code and rebuild the DLL. Nevertheless, a formally released DLL is very common and has a number of advantages:

- Space efficiency - no need to provide all the source code, supporting unit tests, design documentation etc.
- Hiding of intellectual property
- Clients don't need to compile the DLL
- Helps to define formal releases that have passed quality control (i.e. release engineering)
- Developers have very clear boundaries, and more importantly well defined areas of responsibility. When development of a library is shared across hundreds of developers there is a tendency for no-one to take responsibility for quality.

This choice is particularly relevant to very large teams of developers. Should they use solutions that allow all source code to be built by all developers, or should packaging and releasing of DLLs be formalised internally within the team? For very large companies the latter approach must be employed to some degree because having every developer build all the source code isn't scalable.

MSVC `.vcproj` and `.sln` files are influenced by this choice because there is a distinction between explicit linking to a given `.lib` file versus declaring a dependency between projects.

By contrast the Xcpp build system makes this choice orthogonal to the representation of the `.xcpj` and `.xcws` files. Therefore the same files can be used to either build a large solution containing all the source code, or else a smaller solution where some of the projects don't appear, instead using explicit linking to the released `.lib` files.

2.2 A global logical tree

It is common practice for build systems or source code repositories to use a directory named something like "ThirdParty" for where to place external projects. For example, company X may consider that any DLL created by company Y should be placed under "ThirdParty". We reject this notion, because it leads to inconsistency in the logical placement of files, and therefore complexity in the way include and library paths are defined. More importantly it hurts the ability to share and reuse project and workspace files.

We follow the principle that a single logical directory structure holds all source code to be written. This logical structure is defined without regard for the requirements of a particular developer (or even a company). For example, the structure is independent of:

- what projects are physically stored on a given machine;
- what projects are being built and packaged; and
- who is packaging the projects and why.

In this article we use `$` to refer to the one and only root of this logical tree. A *logical path* refers to a path in the logical tree relative to `$`. In order to avoid name clashes a company may want to locate their projects under a directory based on the company name.

2.3 Physical trees

The logical tree is only an abstraction in the sense that it encompasses all projects written at all times by all developers (hoping that name clashes won't occur). Therefore it is only ever possible to physically store some subset of the logical tree on a real computer. We call this a *physical tree*. The root of a physical tree must always correspond to the root `$` of the logical tree. It is convenient to identify a physical tree by the physical location of its root. Let `$1`, `$2`, etc to refer to local paths on a real computer that identify physical locations of the root nodes of physical trees. E.g. a developer may store three distinct physical trees on their computer:

```
$1  =  c:/dev/head
$2  =  c:/dev/experiment
$3  =  d:/devroot/catscan_1_0
```

For a file with absolute path `c:/dev/head/x/y/z`, we regard this path as the concatenation of an absolute path to a physical tree (i.e. `$1 = c:/dev/head`) and a logical path (i.e. `x/y/z` which is the path relative to the root of the physical tree). Therefore we say that `x/y/z` is the file's logical path.

It is allowed (but not required) that each of these physical trees be working sets associated with particular branches or tagged versions stored within particular source code repositories. As such it is typically possible to independently commit/update from these distinct physical trees.

Note that source code repository systems do not normally allow for "mounting" parts of physical trees of one repository into the physical tree associated with another repository (especially when it involves a different vendor). In general it's appropriate to assume that working sets checked out of source code repositories represent distinct trees of directories and files on a developer machine.

Generally it is expected that a source code repository stores an entire physical tree (i.e. that includes `$`), rather than only some sub-tree strictly below `$`. One reason for doing so is a practical one: If we always store a physical tree rooted at `$` in a repository and in a working set we can always check out, commit changes, update, and tag the whole physical tree in a single, convenient and atomic operation.

2.4 Virtual Tree

In this section we define a concept of taking the union of an *ordered* sequence of physical trees. We call this a *virtual tree* because the build system doesn't materialise it. The concept of a virtual tree provides the basis for combining together source code from different sources in a well defined manner, without the need for specialised support from the file system (such as symbolic links to mount directories).

For example, `[$1,$2,$3]` denotes a virtual tree which contains the union of all the directories and files from `$1`, `$2` and `$3`. In the case where there are multiple candidate files with the same logical path, the winner is the file in the physical tree that appears first in the ordered sequence of physical trees. Note that there are some obvious similarities with the ordered sequence of *additional include paths* used by C/C++ compilers. E.g. if `$2` and `$3` (but not `$1`) store a file with the logical path `CatScanner/Image.h` then only the file under `$2` is the one that is said to exist in the virtual tree, because files in `$2` always take precedence over corresponding files from `$3`.

This approach is elegant because a virtual tree is represented very simply as an ordered list of paths, without the need to specify locations for where to logically mount one physical tree within another. There is a tiny overhead in that source code must physically appear in context under appropriate containing directories (because the full logical path must be physically realised). However it is argued that making that context physically explicit helps to document the intention more easily than adhoc alternatives.

An MSVC solution is calculated by providing both the following:

1. A virtual tree (specified with an ordered list of paths); and
2. The logical path to a Xcpp workspace (`.xcws`) file.

The given virtual tree designates the *input* to the build process for the solution. All Xcpp workspace and project (`.xcws`, `.xcpj`) files exist in this virtual tree and therefore can be uniquely identified by a logical path. Workspace files always reference project files (and nested workspace files) using logical paths. Project files always reference subprojects using logical paths. Physically however it means for example that a workspace file on drive d: could end up referencing a project on drive e:

2.5 Dealing with versioning of DLLs

Consider that we want to store four distinct exported versions of the CadStar project on our computer. No problem! We simply use four distinct physical trees. E.g.

```
$1 = c:/dev/CadStar/V1.0
$2 = c:/dev/CadStar/V1.1
$3 = c:/dev/CadStar/V1.2
$4 = e:/dev/CadStar/V2.0
```

These are physical paths to physical trees and therefore the convention for organising these directories on the hard-disk or in a given repository is outside the scope of this article. Some users for example may simply use a large flat list under a single directory. Other users will want to organise the physical trees in a structure using directories. It is of course permissible (and in fact recommended) for this structure to be recorded in a source code repository, i.e. so development teams share the same structure).

A virtual tree references a given physical tree using a local physical path. It follows that any number of distinct virtual trees can reference the same physical tree. In other words the virtual tree idea promotes sharing. For example, V1.2 of CadStar only needs to be stored once on a given development machine, and yet it can be used by many different build environments.

Often a given physical tree holds some exported projects. The nice feature is that the full directory structures are always recorded relative to `$`, which eliminates the need for the developer to manually specify where physical trees need to be "mounted". This largely automates the process of bringing a wide range and disparate set of third party software libraries together to form a build environment. All the user needs to do is specify a virtual tree - i.e. an ordered sequence of local paths to physical trees.

Xcpp project and workspace files are written in a way that is completely independent of versioning concerns. i.e. we never see anything like a version number in either type of file. This helps to ensure that different developers can build the same project in different ways, and not be "fighting" with each other when editing `.xcpj` or `.xcws` files.

A Xcpp workspace can define what projects to package, but doesn't try to stipulate what versions of what projects to package. Version information is instead managed by the developer when specifying the virtual tree to be used for a given build environment.

2.6 Multiple repositories can hold the same exports

Only one entity (e.g. a company or a developer) will formally export version X of library Y. Therefore everyone in the world should be able to agree on what that export actually is. It is therefore appropriate to check-in a given export into any number of distinct source code repositories. It shouldn't create confusion because they should all be identical. This is important to allow for a given repository to be self sufficient or autonomous (e.g. to allow developers to build software despite only having access to a single repository).

2.7 Build and export directories

In order to build an MSVC solution the following must be given:

1. The logical path of the `.xcws` workspace file
2. A virtual tree (defined by an ordered list of paths to physical trees), treated as the *input* for the build process.
3. The path to the *build directory* (used for all generated intermediate files that are output by the build process)
4. The path to the *export directory*, used to store the useful output of the build process.

This information on a given development machine corresponds to a particular *build environment*.

A virtual tree represents the input to the build process. The output of the build process appears under the build and export directories. To avoid any confusion the output of the build is written to a completely separate area which doesn't overlap with the input. i.e. it is assumed that the build and export directories are not contained within any of the physical trees specified in the input virtual tree.

For a given solution, the output of the build is written to either the build directory to hold the intermediate build files, or else the export directory to hold the output files of the build that are packaged in a formal release.

The build directory contains intermediate build files such as `.obj` and `.pch` files. To avoid name clashes `.obj` files are organised under directories according to project name, platform and configuration. It is assumed that filenames are unique within a given project but not across projects.

Since the `.vcproj` and `.sln` files are calculated they are also treated as intermediate files to be stored in the build directory.

2.8 Chained builds

A vital concept is that the output of one build process can serve as part of the input to another. The export directory is organised as a physical tree that can be used to help define another virtual tree, such that the exported libraries are available as inputs for another MSVC solution. It follows that paths down from the export directory can be regarded as logical paths.

When a `.xcws` file is used to generate MSVC `.vcproj` and `.sln` files, the result depends on whether original or exported versions of projects are found within the input virtual tree. The virtual tree is not defined by the `.xcws` file, so the same `.xcws` file can generate quite different development environments to suit the needs of the individual developer.

When a project is exported, a `.xcpj` file is written to the export directory with the same logical path as the original `.xcpj` file. This file resembles the original except it only lists the public files of the project (this is achieved by simply assuming that everything under a directory named `src` isn't public and therefore isn't exported). It also inserts the keyword `export` into the exported version of the `.xcpj` file to flag it as exported.

When generating an MSVC solution that links against an exported library, it is possible to generate a utility `.vcproj` project file that conveniently allows the (read only) public header files to be viewed by the developer. This option is available through the `solutionContainsPrepackagedProjects` flag that can be assigned in a `.xcpp` configuration file (see section 5.1).

2.9 Public header files

The Xcpp Build System sets up additional includes for the compiler that match the sequence of paths to the physical trees in the input virtual tree. When a programmer uses a `#include` directive to a public header file in a different project, a logical path to the header file is recommended. This will always bind to the appropriate file in the virtual tree.

Libraries typically define one or more public header files that are regarded as part of the output of the build to be exported. The Xcpp build system supports the copying of public header files from the input virtual tree to the output export directory. This is done without changing the relative path from the root of the virtual tree. Therefore public header files in the export directory have the same logical path as they do in the input virtual tree. This allows dependent projects to be build either against the original source code or else the packaged version with no need to worry about additional includes.

3 Xcpp Project Files

An Xcpp project file typically has extension `xcpj`, and is the counterpart to a MSVC `.vcproj` file. It specifies all the compiler and linker command line options and the set of project files to be compiled and linked to build a single target. Unlike a `.vcproj` file, a `.xcpj` file is intended to be directly edited by

the user. A .vcproj file can be automatically generated from a given .xcpj file. The grammar is presented in the appendix.

A lexical scanner similar to the one used for the C programming language is used. It allows the use of //... or /*...*/ comments throughout the file. White space between tokens is not generally significant.

In general paths within .xcpj and .xcws files must use forward slashes (not backslashes). It is an error to include a leading or trailing slash in a path.

Every .xcpj file is comprised of the following three sections:

1. The definitions of $TARGET_TYPE and $ROOT_TO_PROJDIR
2. Specification of the configurations, the compiler and linker switches for each configuration across the project, and the sub-projects. The following directives can appear (any number of times and in any order): @import, config{...}, +cpp{...}, -cpp{...}, +rc{...}, -rc{...}, +link{...}, -link{...}, subproj{...}, directories, prebuildevent{...}, prelinkevent{...}, postbuildevent{...}, toolFiles{...}, general{...}.
3. The set of files in the project, in the form of a tree structure using curly braces to represent nested directories. See section 3.13.

3.1 Target types

There are five possible types of target for a given project. The target type is represented in a variable named $TARGET_TYPE which must be defined at the beginning of an .xcpj file and which can take on one of the following double quoted string values:

```
"Application"
"Static Library"
"Console Application"
"Dynamic-Link Library"
"Utility Project"
```

The predefined variable $TGT_EXTENSION is calculated automatically from the specified target type. This takes on the value "exe" for an application, "lib" for a static library or "dll" for a dynamic link library.

3.2 Project directory

It is assumed every project is uniquely identified by the logical path to its *project directory*. Normally all the files in a project live in the tree rooted at the project directory. Immediately after the definition of $TARGET_TYPE, an .xcpj file must define the variable $ROOT_TO_PROJDIR which specifies the logical path to the project directory. A predefined variable named $PROJDIR_TO_ROOT is automatically calculated as the inverse of $ROOT_TO_PROJDIR.

By convention the name of the project directory is regarded as the *project name*. It is not assumed that project names are globally unique. However this assumption is made within a given solution. The project name is implied from the value of $ROOT_TO_PROJDIR (i.e. it is always the last directory name in this path). The project name is automatically made available in a variable named $PROJNAME.

Files that are not exported are conventionally stored under a directory named src under the project directory.

For example if $PROJDIR_TO_ROOT="x/y/z" then $PROJNAME="z" and $PROJDIR_TO_ROOT="../../..". Also it is assumed that the associated .xcpj file will have logical path x/y/z/src/z.xcpj. This convention is used for all projects.

3.3 Using the Xcpp Macro Preprocessor

The *Xcpp Macro Preprocessor* [1] is applied to the .xcpj file before it is parsed. For example it is possible to declare macros of the form

```
@def name = substitution-string
```

The preprocessor also provides @if-@else directives, making it possible to conditionally define configurations, compiler switches etc. Note however that it is not expected that the preprocessor needs to be used very often, if at all.

3.4 Variables

A variable is defined like this

```
$name = "my string value"
```

Variables can be defined in both .xcpj and .xcpp files. A variable can be reassigned a new value. Currently variables can only be "invoked" from within string literals (meaning that the variable reference of the form $(name) is replaced with its current value within the string literal, rather like a macro substitution). Note that the brackets are used in a reference but not a definition of a variable. E.g.

```
+link
{
  /OUT:"$(EXPORT)/$(PLATFORM)/$(CONFIG)/
    $(PROJNAME).$(TGT_EXTENSION)"
}
```

The variables defined in the .xcpp file (but not the .xcpj file itself) are made available as macros to the Xcpp Macro Preprocessor by enclosing the variable name in underscores. E.g. there are macros named _EXPORT_ and _PLATFORM_.

3.4.1 Predefined variables

The following variables are predefined:

- $(PROJNAME) - the name of the project.
- $(EXPORT) - the path to the export directory relative to the build directory.
- $(TARGET_TYPE) - The type of target to be built
- $(TGT_EXTENSION) - Either "exe", "dll" or "lib" depending on $TARGET_TYPE.
- $(ROOT_TO_PROJDIR) - The path from the virtual tree root to the project directory.
- $(PROJDIR_TO_ROOT) - The path from the project directory to the virtual tree root.
- $(PLATFORM) - The name of the platform. E.g. "Win32" or "Pocket PC 2003 (ARMV4)"
- $(CONFIG) - The name of the configuration - typically either "Release" or "Debug".

3.5 Example of a .xcpj file

The following Xcpp project file (.xcpj), located at Ceda/cxPython/src/cxPython.xcpj is used to build cxPython.dll:

```
$TARGET_TYPE = "Dynamic-Link Library"
$ROOT_TO_PROJDIR = "Ceda/Core/cxPython"
xcpp

@import "Ceda/BaseDefaults.xcpjh"
@import "Ceda/LinkPython.xcpjh"

subproj
{
  "Ceda/Core/cxUtils"
  "Ceda/Core/cxObject"
}

{
  "src"
  {
    "AssignFromPyObject.cpp"
    "BootStrapCedaModule.cpp"
    "BoxInt64.cpp"
    "BoxInt64.h"
    "Utility.cpp"
    "Utility.h"
    "WrapEnum.cpp"
    "WrapNameSpace.cpp"
    "WrapClass.cpp"
    "WrapGlobalFunction.cpp"
    "WrapPointerValue.cpp"
    "WrapArrayVar.cpp"
    "WrapVariable.cpp"
    "WrapCedaBuiltInType.cpp"
    "WrapClassVariable.cpp"
    "WrapVector.cpp"
    "WrapAttributeOnInterfacePtr.cpp"
    "WrapInterfacePtr.cpp"
    "WrapMethodOnInterfacePtr.cpp"
    "WrapMethodOnClassVariable.cpp"
    "WrapModelStructVar.cpp"
    "WrapModelArrayVar.cpp"
    "WrapModelVectorVar.cpp"
    "cxPython.cpp"
    "cxPython.rc"
    "Resource.h"
    "StdAfx.cpp" : +cpp { /Yc"StdAfx.h" }
    "StdAfx.h"
  }
  "cxPython.h"
}
```

3.6 Importing files

Normally a project imports the configurations and the compiler and linker switches from another file, avoiding the need to repeat these settings in every project.

An imported file can in turn import other files and so on indefinitely (as long as there is no cycle). An imported file may define macros and variables that are visible from the file that issues the import directive. Also an imported file can define configurations, apply compiler and linker switches and add subprojects.

An imported file normally only represents a part of a project file, and the convention is to use the extension `xcpjh` instead of `xcpj`.

Typically a logical path to the file to imported is used. The file will be searched in the virtual tree accordingly. Alternatively a path that is relative to the physical location of the file containing the `@import` directive may be used.

3.7 Configuration set

The following is an example of how to define the set of configurations:

```
config
{
  "Debug|Win32" ["MBCS"]
  "Release|Win32" ["MBCS"]
  "Debug Unicode|Win32" ["Debug" "Unicode"]
  "Release Unicode|Win32" ["Release" "Unicode"]
  "Debug|Pocket PC 2003 (ARMV4)" ["WinCE"]
  "Release|Pocket PC 2003 (ARMV4)" ["WinCE"]
  "Debug|Smartphone 2003 (ARMV4)" ["WinCE"]
  "Release|Smartphone 2003 (ARMV4)" ["WinCE"]
}
```

Each entry is of the form `"config|platform"` plus optionally a list of any number of white space delimited tags in square brackets. In the above case there are 6 configurations and 3 platforms. The tag `"WinCE"` has been applied to the last 4 configurations.

For a given entry of the form `"config|platform"`, the config is available in a variable named `$(CONFIG)` and the platform in a variable named `$(PLATFORM)`. E.g. the platform and config names can be used to specify the output file of the linker:

```
+link
{
  /OUT:"$(EXPORT)/$(PLATFORM)/$(CONFIG)/
    $(PROJNAME).$(TGT_EXTENSION)"
}
```

A config block can be repeated to add more and more configurations. This can be useful in a `@import` file which specifies a base set of configurations, yet allows projects to add additional configurations.

Unfortunately in both versions of MSVC that were tried (VC 2005 and VC 2008) the IDE allows the user to select configurations that were never defined, such as

```
  Debug Unicode|Pocket PC 2003 (ARMV4)
```

VC even allows the user to try to build it! Not surprisingly the build fails (e.g. because the additional includes aren't even defined).

Note that a configuration that targets Win32 *must* name the platform `"Win32"` or else MSVC gets upset. It is unknown why this is the case given that every imaginable setting is specified explicitly in the `.vcproj` file, raising the question of why the platform name is significant.

3.8 Configuration lists

Compiler and linker switches can be applied to particular configurations, expressed using combinations of the following:

- Configuration names (such as `"Debug"` or `"Release"`)
- Platform names (such as `"Win32"` or `"Pocket PC 2003 (ARMV4)"`
- Configuration tags (such as `"WinCE"`, `"MBCS"` or `"Unicode"`)
- Compiler version names (such as `"VC8"` or `"VC9"`)
- Target types (i.e. values of `$TARGET_TYPE` such as `"Application"` or `"Dynamic-Link Library"`)

As an example, the following switches are applied to all configurations named `"Release"` (i.e. irrespective of the target type, platform and compiler)

```
+cpp("Release")
{
  /Ot
  /FD
  /D "NDEBUG"
}

+link("Release")
{
  /PROFILE
  /MAP:"$(EXPORT)/$(PLATFORM)/$(CONFIG)/
```

```
    $(PROJNAME).map"
  /INCREMENTAL:NO
  /OPT:
  /OPT:ICF
}
```

Even though there is some risk of confusion to a C/C++ programmer, `'|'` denotes a logical ANDing. This is done because Microsoft have set a precedent for using the syntax `config|platform` for a particular configuration on a particular platform.

E.g the following switch is applied to the configuration `"Release"` for platform `"Win32"`

```
+cpp("Release|Win32")
{
  /MD
}
```

The order doesn't matter - the following works just as well:

```
+cpp("Win32|Release")
{
  /MD
}
```

In fact any of the specifiers can be ANDed in this way. E.g. we can use a tag name:

```
+cpp("Release|WinCE")
{
  /MT
}
```

A comma delimited list of strings is used for logical ORing. E.g.

```
+cpp(
  "Debug|WinCE|Console Application|VC9",
  "Debug|Win32|Application")
{
  /MTd
}
```

3.9 C++ Compiler switches

Compiler switches can be incrementally added (using `+cpp`) or removed (with `-cpp`) in a top to bottom reading of the `.xcpj` file. For example:

```
+cpp
{
  /nologo /W3 /Zm500 /Zi /EHsc
  /D "_CRT_SECURE_NO_DEPRECATE"
  /Yu"StdAfx.h"
  /Fp"./$(PLATFORM)/$(CONFIG)/$(PROJNAME)/
    $(PROJNAME).pch"
  /Fo"./$(PLATFORM)/$(CONFIG)/$(PROJNAME)/"
  /Fd"./$(PLATFORM)/$(CONFIG)/$(PROJNAME)/"
}

+cpp("Application", "Static Library",
    "Dynamic-Link Library")
{
  /D "_WINDOWS"
}

+cpp("Console Application")
{
  /D "_CONSOLE"
}

+cpp("Static Library", "Dynamic-Link Library")
{
  /D "_WINDLL"
  /D "$(PROJNAME)_EXPORTS"
}

+cpp("MBCS")
{
  /D "_MBCS"
}

+cpp("Unicode", "WinCE")
{
  /D "_UNICODE" /D "UNICODE"
}

+cpp("Win32")
{
  /fp:precise /D "WIN32"
}

+cpp("WinCE")
{
  /Os /GR /fp:fast
  /D "_WIN32_WCE=0x420"
  /D "UNDER_CE"
  /D "WINCE"
  /D "WINDOWSCE_DLL_EXPORTS"
  /D "ARM"
  /D "_ARM_"
  /D "_WINDLL"
}

+cpp("Pocket PC 2003 (ARMV4)")
{
  /D "WIN32_PLATFORM_PSPC"
}

+cpp("Smartphone 2003 (ARMV4)")
{
  /D "WIN32_PLATFORM_WFSP"
}

+cpp("Release")
{
  /Ot /FD /D "NDEBUG"
}

+cpp("Debug")
{
  /Od /Gm /RTC1
  /D "_DEBUG" /D "DEBUG"
}

+cpp("Release|Win32") { /MD }
+cpp("Debug|Win32") { /MDd }
+cpp("Release|WinCE") { /MT }
+cpp("Debug|WinCE") { /MTd }
```

The compiler switches are additive, making it easy to put common switches in a imported file.

Note the following:

- Compiler switches can be added across all configurations or else a given subset of the configurations.
- Variables can be specified in the form `$(name)` within the strings and they are automatically expanded like a macro.
- When specifying relative paths (e.g. for the switches `/I`, `/Fp`, `/Fo` and `/Fd`), it is assumed that paths are absolute or else relative to the location of the generated `.vcproj` file, which is the build directory.
- `+cpp {...}`, `-cpp {...}` can be used to add or remove compiler switches to all files in the project, or else to individual files in the project.
- There is support for per file per configuration compiler switches. See section 3.13.12.

3.9.1 C++ additional include paths

For each project it's implicit that the following are in the additional include paths:

- The project directory
- The `src` directory under the project directory

Additional directories can be added to the include paths using the `/I` switch. For example:

```
+cpp
{
  /I "$(VIRTUAL_TREE)"
  /I "c:/dev/boost"
}
```

A path is absolute or else relative to the build directory. If a path begins with `$(VIRTUAL_TREE)`, then a special kind of macro substitution is performed, where the whole path is repeated such that the prefix `$(VIRTUAL_TREE)` is replaced by the path to each physical root specified in the virtual tree in turn.

3.10 Resource compiler switches

`+rc {...}` and `-rc {...}` blocks are used to define the resource compiler switches. For example:

```
+rc
{
  /d "_DEBUG"
  /l 0xc09 /x /v
  /I "thirdparty/gadgets/include"

}

+rc("WinCE")
{
  /d "_WIN32_WCE"
  /d "$(CEVER)"
  /d "UNDER_CE"
  /d "$(PLATFORMDEFINES)"
}
```

3.11 Linker switches

The following shows an example of specifying the linker command line options

```
+link
{
  /OUT:"$(EXPORT)/$(PLATFORM)/$(CONFIG)/
    $(PROJNAME).$(TGT_EXTENSION)"
  /PDB:"$(EXPORT)/$(PLATFORM)/$(CONFIG)/
    $(PROJNAME).pdb"
  /LARGEADDRESSAWARE
  /DEBUG
  /NOLOGO
}

+link("Dynamic-Link Library")
{
  /DLL
  /IMPLIB:"$(EXPORT)/$(PLATFORM)/$(CONFIG)/
    $(PROJNAME)/$(PROJNAME).lib"
}

+link("Release")
{
  /PROFILE
  /MAP:"$(EXPORT)/$(PLATFORM)/$(CONFIG)/
    $(PROJNAME).map"
  /INCREMENTAL:NO
  /OPT:REF /OPT:ICF
}

+link("Debug")
{
  /INCREMENTAL
}

+link("Console Application")
{
  /SUBSYSTEM:CONSOLE
}

+link("Application|Win32")
{
  /SUBSYSTEM:WINDOWS
}

+link("Win32")
{
  /MACHINE:X86
  /MANIFEST
  /MANIFESTFILE:"$(PLATFORM)/$(CONFIG)/
    $(PROJNAME)/$(PROJNAME).$(TGT_EXTENSION)
    .intermediate.manifest"
  "kernel32.lib"
  "user32.lib"
  "gdi32.lib"
  "winspool.lib"
  "comdlg32.lib"
  "advapi32.lib"
  "shell32.lib"
  "ole32.lib"
  "oleaut32.lib"
  "uuid.lib"
  "odbc32.lib"
  "odbccp32.lib"
}

+link("WinCE")
{
  /SUBSYSTEM:WINDOWSCE,4.20
  /MACHINE:ARM
  /ARMPADCODE
  /MANIFEST:NO
  /NODEFAULTLIB:"oldnames.lib"
  /STACK:65536,4096
  "coredll.lib"
  "corelibc.lib"
  "ole32.lib"
  "oleaut32.lib"
  "uuid.lib"
  "commctrl.lib"
}
```

Note the following:

- Libraries must be specified in double quotes
- The linker should be regarded as running from the location of the generated `.vcproj` file - i.e. the build directory.
- Per configuration linker switches can easily be specified (in the same fashion as for compiler switches)
- Obviously there is no concept of per file linker switches.

3.12 Sub-projects

A *subproject* is a project on which the current project depends (i.e. there is a linking dependency). When a project specifies a subproject, the logical path to the project directory is given for the subproject. For example:

```
subproj
{
  "Ceda/Core/cxUtils"
  "Ceda/Core/cxObject"
}
```

Note the following:

- Sub-projects are specified in an additive fashion. i.e. each `subproj{ ... }` directive adds to the overall set of sub-projects.
- Per configuration sub-projects are not supported

- Each sub-project must be specified using a logical path to the project directory in double quotes. Forward slashes must be used.

3.13 Input files

A tree structure specifies all the files in the project. File and directory names must be enclosed in double quotes. A string represents a directory name if and only if it is followed by a left brace. It is conventional (but not required) to list directories before files.

```
{
  "dir1"
  {
    "dir2"
    {
      "file1.cpp"
      "file1.h"
      "file2.cpp"
      "file2.h"
    }
    "dir3"
    {
      "dir4"
      {
        "file4.cpp"
        "file4.h"
      }
      "file3.cpp"
      "file3.h"
    }
  }
  "file5.cpp"
  "file5.h"
}
```

In this case the directory structure on disk is reflected in the folder structure presented in MSVC, using folder icons for directories `dir1`, `dir2`, `dir3` and `dir4`. We will see below that it is possible to break this direct correspondence, if it's appropriate.

The outermost curly braces is always assumed to enclose the content of the project directory (see section 3.2).

3.13.1 Wild cards

The Xcpp build system supports wild cards when specifying files/directories to add to a project. Consider the following specification of the files in a project:

```
{
  "src"
  {
    "ArchiveTests.cpp"
    "CacheMapTests.cpp"
    "CRCTests.cpp"
    "FileTests.cpp"
    "GuidTests.cpp"
    "HeapAllocTests.cpp"
    "HexTests.cpp"
    "IndentingStringStreamTests.cpp"
    "Lessons.cpp"
    "MD5Tests.cpp"
    "MultiSubStringTests.cpp"
    "PagedBufferTests.cpp"
    "ParserDemo.cpp"
    "PseudoRandomTests.cpp"
    "SessionValueCacheTests.cpp"
    "ThreadTests.cpp"
    "RefCounter.cpp"
    "VariableLengthSerialiseTests.cpp"
    "VectorTests.cpp"
    "VectorTests.h"
    "xstringTests.cpp"
    "xostreamTests.cpp"
    "txUtils.cpp"
    "StdAfx.cpp" : +cpp { /Yc"StdAfx.h" }
    "StdAfx.h"
  }
}
```

An alternative is for the files in the project to be determined automatically from the file system:

```
{
  "src"
  {
    "StdAfx.cpp" : +cpp { /Yc"StdAfx.h" }
    "*.cpp"
    "*.h"
  }
}
```

Wild card searching is performed against the virtual tree, and therefore may involve searching through multiple physical trees on a development machine.

Note that `StdAfx.cpp` isn't added twice. In general, wild card searches never add a folder or file with the same logical path more than once.

3.13.2 Name Patterns

We refer to a string such as `"*.cpp"` as a *name pattern*. A name pattern cannot contain any slash characters. Most generally name patterns can be used to filter either file names or directory names.

For the file named `"foo.bar"`, the following name patterns all successfully match the file name:

```
*
f*
*.*
*.b*
foo.bar*
f*.bar
f*.*r
f??.bar
???.???
f*.b?r
*?oo.bar
```

whereas the following name patterns fail:

```
*?foo.bar
f?.bar
g*
*.exe
```

3.13.3 Name filters

A *name filter* is either a prescriptive or proscriptive set of name patterns,

A name pattern is represented using a double quoted string. A name filter consists optionally of a '+' or '-', then either a single pattern or else a list of zero or more white space delimited patterns in square brackets. An initial '-' means the specification is proscriptive (rather than prescriptive, which is the default). With the '-' a name is rejected if it matches any of the patterns, whereas with '+' a name is accepted if it matches any of the patterns.

The name filter `-[]` means reject nothing, or in other words accept anything, and is equivalent to `"*"`.

Note that the following four lines are all equivalent name filters:

```
"*.cpp"
["*.cpp"]
+"*.cpp"
+["*.cpp"]
```

3.13.4 Recursive search

There is support for recursion through directories by appending an asterisk after the file name filter. E.g.

```
{
  "dir1"
  {
    "dir2"
    {
      ["*.cpp" "*.h"] *
    }
  }
}
```

In this case all `.cpp` and `.h` files under `dir2` or any of its subdirectories recursively are added to the project. This could cause many folders to be shown under `dir2` in MSVC. Note that only non-empty directories are added to the `.vcproj` file and therefore appear in MSVC.

3.13.5 Flat recursive search

Recursion through directories can have the structure flattened as it appears in MSVC. This is easily achieved by prefixing the asterisk with the keyword `flat` as follows:

```
{
  "dir1"
  {
    "dir2"
    {
      ["*.cpp" "*.h"] flat *
    }
  }
}
```

Now all the `.cpp` and `.h` files are displayed in MSVC in a single flat list under `dir2`.

3.13.6 Rename a folder in MSVC

It is possible to rename a folder as it appears in MSVC. For example:

```
{
  "dir1"
  {
    "dir2" as "x"
    {
      ["*.cpp" "*.h"] flat *
    }
  }
}
```

Now directory `dir2` will appear as a folder named `x` in MSVC.

3.13.7 Using empty strings

`as` doesn't just allow for renaming. It can also allow a folder to be added to MSVC where there is no corresponding directory in the virtual tree, and vice versa. This is achieved by using empty strings on either side of `as`. E.g.

```
{
  "" as "Header"
  {
    "*.h" flat *
  }
  "" as "Source"
  {
    "*.cpp" flat *
  }
}
```

In this case there are folders in MSVC named `Header` and `Source` but no corresponding directories in the virtual tree.

Conversely, in the following example `dir2` has been aliased to an empty string. A flattening affect is achieved, in that all the `.h` files appear directly under `dir1` instead of `dir2`.

```
{
  "dir1"
  {
    "dir2" as ""
    {
      "*.h"
    }
  }
}
```

Use the following mnemonic to help understand this:

```
"virtual-tree-directory-name" as
    "MSVC-folder-name"
```

3.13.8 Directory names containing ".."

It is permissible for directory names to contain `'..'` in the manner illustrated below to step upwards out of the project directory. For example:

```
$TARGET_TYPE = "Utility Project"
$ROOT_TO_PROJDIR = "x/y/z"
{
  "../../.." {"*.xcws"*}  // $
  "../.." {}              // x
  ".." {}                 // y
  "" {}                   // z
}
```

`"../../.."` is a right inverse of `$ROOT_TO_PROJDIR` and therefore represents the root of the virtual tree. It is displayed as a folder named `$` within MSVC. The pattern `"*.xcws"` is applied recursively to the virtual tree under `$`, so we end up with every Xcpp workspace file in the entire virtual tree.

Using `"../.."` denotes the directory named `x`, and would be displayed as a folder named `x` in MSVC.

There is a predefined variable named `$PROJDIR_TO_ROOT` which is calculated as a right inverse of `$ROOT_TO_PROJDIR`. Therefore another way to show all the workspaces in the virtual tree is:

```
$TARGET_TYPE = "Utility Project"
$ROOT_TO_PROJDIR = "x/y/z"
{
  "$(PROJDIR_TO_ROOT)" as "Workspaces"
  {
    "*.xcws" flat *
  }
}
```

In this case the folder appearing in MSVC has been renamed `Workspaces`, and the `.xcws` files in the folder are displayed in a flat list.

3.13.9 Example using macro preprocessor

Here is a more complex example using the xcpp macro preprocessor:

```
$TARGET_TYPE = "Utility Project"
$ROOT_TO_PROJDIR = "Ceda/CedaExtras"
@import "Ceda/BaseDefaults.xcpjh"
directories  -".svn"
@def mProjFiles = ["*.xcpj" "*.xcws" "*.xcpjh"]
    flat *
@def mProjectFilesInDirs(L) = @for(dir in L) dir
    { mProjFiles }
@def ROOT = "$(PROJDIR_TO_ROOT)"
{
```

```
  ROOT as "Workspaces"
  {
    +"*.xcws" flat*
  }
  ROOT
  {
    "Ceda"
    {
      mProjectFilesInDirs(["App", "App1",
          "Build", "CedaExtras", "Core",
          "Core1"])
      "_BUILD"
      {
        -["*.exe" "*.dll"] *
      }
    }
      mProjFiles
  }
}
```

Note that since all the `.xcws` files under `$` are placed under Workspaces, they cannot appear anywhere else. In general a file will only show up at most once in a given project - i.e. where it first appears in a top to bottom reading of the `.xcpj` file.

3.13.10 Directories filter

While processing the `.xcpj` file from top to bottom, there is a global variable that records a *Directories Name Filter* (DNF) to be used when recursing into nested directories. Recursion only proceeds into those directories having a name that complies with the currently set DNF.

The DNF is assigned using the keyword `directories` followed by a name filter following the syntax defined in section 3.13.3.

In the following example the recurse under `dir3` will not proceed into any directory named `bin`, whereas the recurse under `dir4` will only proceed into directories that begin with 's' or 'd'.

```
{
  "dir1"
  {
    "dir2"
    {
      directories -"bin"
      "dir3" {-[]*}

      directories +["s*" "d*"]
      "dir4" {-[]*}
    }
  }
}
```

Initially the DNF allows for all directories, which is equalavent to `-[]` (which is also equivalent to `+[*]` or just `[*]` since the `+` is optional).

When the source code is checked out from a SubVersion repository each directory in the working set contains internal files stored under a subdirectory named `.svn`. It could be thought that the following should be used to avoid pulling these hidden files and directories into the project:

```
directories  -".svn"
```

However this is not actually required because the Xcpp build system automatically ignores directories with the 'hidden' attribute set.

3.13.11 Absolute paths

It is possible to use absolute paths (i.e. paths that begin with a slash, or begin with `x:/` where `x` denotes a drive letter). E.g.

```
$MSVC = "C:/Program Files/Microsoft Visual
    Studio 8"
{
  "$(MSVC)/VC/include" as "msvc"
  {
    "*.h" *
  }
}
```

3.13.12 Applying per file per config compiler switches

Following a file or file filter (even a recursive one), a colon then a comma separated list of items can be specified. For C++ files each item is one of the following:

- `+cpp{...}` – to add additional compile switches to the files for every configuration
- `-cpp{...}` – to remove compile switches for the files for every configuration
- `+cpp(configs){...}` – to add additional compile switches to the files for the given configurations
- `-cpp(configs){...}` – to remove compile switches for the files for the given configurations
- `exclude` – to exclude the files from the build for every configuration

```
  "MyGroup"
  {
    "*.cpp" flat * : +cpp{/D "BLAH"}
    "g*.cpp" * : -cpp("Release"){/D "BLAH"},
        +cpp{/O1}
    "f*.cpp" : exclude
  }
```

Note that since file filters can overlap (in the set of files that they reference), for a given file and config the effect is roughly to take a union of various settings. More precisely for a given file and config there is a sequence of `+cpp{...}` and `-cpp{...}` that are applied in order in a top to bottom reading of the `.xcpj` file to end up with a particular result.

For resource files, `+rc{...}` or `-rc{...}` items are applicable instead.

4 Xcpp Workspace Files

An *Xcpp Workspace File* typically uses the extension `xcws`. It defines a set of projects to be built (i.e. added to an MSVC solution). A workspace file is stored in the logical tree and therefore is assumed to be uniquely identified by a logical path. There are no additional constraints on the possible location of a workspace file.

Some projects are added to a workspace indirectly. For example, when adding a project, all its subprojects are implicitly added as well. These subprojects may in turn have subprojects and so on that are also added implicitly. Note that a project is only added to a MSVC solution once even though it may be a subproject of many projects.

One "parent" workspace can reference another "child" workspace. The parent inherits all the projects defined by the child. This referencing between workspace files can form arbitrary DAG structures (i.e. cycles are not permitted). The inheritance of projects is transitive.

The file format of an `xcws` consists of a name (which will be used for the generated MSVC solution file), followed by a list of double quoted strings enclosed in curly braces. Each string is a logical path using only forward slashes, either to a project directory or else to a child `.xcws` file. E.g.

```
"Object"
{
  "Ceda/Core/Utils.xcws"
  "Ceda/Core/cxObject"
  "Ceda/Core/Object/exObject"
  "Ceda/Core/Object/txObject"
}
```

4.1 Additional dependencies

Projects can be nested in the `.xcws` file to create additional dependencies between projects when the MSVC `.sln` file is generated.

A workspace can reference another workspace in order to include all the projects and dependencies specified by that workspace. This results in the union of all the project dependencies.

5 xcpp.exe

`xcpp.exe` is a console application able to generate the MSVC solution and project files in the manner described in this article. The settings are provided in a *xcpp configuration file*. By convention this file has extension `xcpp`. The format of this configuration file is described below in section 5.1.

On the command line, `xcpp.exe` takes the path to the configuration file. Any extra command arguments act as though they are appended to the end of the configuration file. This allows some of the configuration settings to be specified on the command line if that is convenient.

5.1 Xcpp configuration file

The `.xcpp` file format mostly consists of settings that take the form `name = value`. Names can be reassigned. In particular this includes the definition of variables, following the same syntax as in `.xcpj` files (see section 3.4). In fact these variables are made available during the subsequent processing of each of the `.xcpj` files. This provides an ideal mechanism to pass information through to every project. It is particularly useful for passing through include and library paths to third party libraries (like boost).

`xcpp.exe` actually allows for building all the targets by running the C++ compiler, resource compiler and the linker itself. To allow this it must be provided with PATH, LIB and INCLUDE variables. The values of these variables must be white space delimited strings enclosed in curly braces.

The compiler can be set to `vc6`, `vc8` or `vc9`. This affects the generation of the `.vcproj` and `.sln` files for MSVC.

The variable `virtualTree` is assigned with an ordered list of paths to the roots of the physical trees. These paths can be absolute or else relative to the build directory.

The variable `$EXPORT` must be assigned with the path to the export directory. This path can be absolute or else relative to the build directory.

There are a number of boolean flags that can also be set. These are detailed in the example of a `.xcpp` file given below:

```
$MSVS = "c:/Program Files/Microsoft Visual
    Studio 8"

// VC8 professional
PATH =
{
  "$(MSVS)/VC/bin"
  "$(MSVS)/Common7/IDE"
  "$(MSVS)/VC/PlatformSDK/bin"
  "c:/windows"
  "c:/windows/system32"
}
INCLUDE =
{
  "$(MSVS)/VC/include"
  "$(MSVS)/VC/PlatformSDK/include"
  "$(MSVS)/VC/atlmfc/include"
}
LIB =
{
  "$(MSVS)/VC/lib"
  "$(MSVS)/VC/PlatformSDK/lib"
  "$(MSVS)/VC/atlmfc/lib"
}
compiler = vc8

virtualTree =
{
  "c:/dev/head"
  "c:/dev/utils/v2"
  "c:/dev/cad/v1"
}

$EXPORT = "../export"

// Enables writing of more verbose information
// to stdout
diagnostics = false

// Indicates whether to copy public header files
// from the input virtual tree to the output
// directory $EXPORT
exportPublicHeaders = true

// Indicates whether to copy dll and exe files
// from the input virtual tree to the output
// directory $EXPORT.
repackageExecutableTargets = true

// Indicates whether pre-packaged projects in
// the input virtual tree are to appear in the
// MSVC solution.
solutionContainsPrepackagedProjects = true

// Indicates whether public header files copied
// from the input virtual tree to the output
// directory $EXPORT will be made read only.
makeExportedHeaderFilesReadOnly = true

// Generate MSVC .vcproj and .sln files
mode = gen
```

In general, `.xcpp` config files support reassignments of variables. Only the last assigned value of a variable takes effect. This combined with the support for `#include` directives makes it possible to inherit default values and override a small subset of the settings.

5.1.1 Settings relevant to translation

The following settings are relevant to the *translation* mode available to `xcpp.exe`. Translation is outside the scope of this article.

```
// If true causes xcpp to write the files under
// ./source without compiling and linking them
// using the VC compiler and linker.
translateOnly = false

// Forces xcpp to rewrite all files under
// ./source
rebuildAll = false

// Indicates whether to rewrite files under
// ./source that are token equivalent
writeTokenEquivalent = false

// Indicates whether xcpp translated files
```

```
// written under ./source are made read only.
makeTranslatedFilesReadOnly = true
```

A .xcpj grammar

We use [2] to define the EBNF syntax. Let `stringLiteral` stand for a double quoted string literal and `intLiteral` for an integer literal.

The grammar is presented in bottom up order (i.e. there is a tendency to define non-terminals before they are referenced)

A.1 Configurations

```
targetType =
  '"Application"' |
  '"Static Library"' |
  '"Console Application"' |
  '"Dynamic-Link Library"' |
  '"Utility Project"';

compilerName =
  '"vc6"' | '"vc8"' | '"vc9"';

(* E.g. "Debug" *)
configName = stringLiteral;

(* E.g. "Win32" *)
platformName = stringLiteral;

(* E.g. "Debug|Win32" *)
configAndPlatform = stringLiteral;

(* E.g. "WinCE" *)
tagName = stringLiteral;

(* E.g. "Debug|Win32" ["Unicode"] *)
singleConfigDef =
  configAndPlatform,
  [
    '[', {tagName}, ']'
  ];

configsDef =
  'config',
  '{',
    { singleConfigDef },
  '}';

(* E.g. "Debug|WinCE|vc8"
   Each string between the | must be either a
   tagName, configName, platformName,
   compilerName or targetType
*)
configRef = stringLiteral;

(* E.g. ("Debug|WinCE|vc8", "Win32") *)
configRefList =
  '(',
  [configRef, { ',', configRef } ],
  ')';
```

Grammar for configurations

A.2 C++ Compiler switches

```
cppBoolSwitch1 =
  '/GR' | '/GL' | '/GS' | '/Gy' | '/fp:except';

cppBoolSwitch2 =
  '/nologo' | '/Oi' | '/Op' | '/Oy' | '/GT' |
  '/X' | '/C' | '/GF' | '/Gm' | '/RTCc' |
  '/Za' | '/J' | '/openmp' | '/Fx' | '/doc' |
  '/WX' | '/Wp64' | '/showIncludes' | '/u' |
  '/FC' | '/Zl' | '/Zc:wchar_t' |
  '/Zc:forScope' | '/QRinterwork' | '/QRfpe';

cppEnumSwitch =
  '/Od' | '/O1' | '/O2' | '/Ox' |
  '/Ob1' | '/Ob2' |
  '/Ot' | '/Os' |
  '/EHsc' | '/EHa' | '/GX' |
  '/RTCs' | '/RTCu' | '/RTC1' | '/RTCsu' |
    '/GZ' |
  '/MT' | '/MTd' | '/MD' | '/MDd' |
  '/Zp1' | '/Zp2' | '/Zp4' | '/Zp8' | '/Zp16' |
  '/QRarch4' | '/QRarch5' | '/QRarch4t' |
    '/QRarch5t' |
  '/TC' | '/TP' |
  '/arch:SSE' | '/arch:SSE2' |
  '/fp:precise' | '/fp:strict' | '/fp:fast' |
  '/FA' | '/FAcs' | '/FAc' | '/FAs' |
  '/FR' | '/Fr' |
  '/W0' | '/W1' | '/W2' | '/W3' | '/W4' |
  '/Z7' | '/Zi' | '/ZI' |
  '/Gd' | '/Gr' | '/Gz' |
  '/errorReport:prompt' |
    '/errorReport:queue' |
  '/clr' | '/clr:pure' | '/clr:safe' |
    '/clr:noAssembly' | '/clr:oldSyntax';

cppStrArgSwitch =
  '/Fd' | '/Fe' | '/Fo' | '/Fp' | '/FR';

cppListStrArgSwitch =
  '/I' | '/D' | '/FI' | '/FU' | '/U';

cppPchSwitch =
  '/Yc' | '/Yu' | '/YX';

cppCompilerSwitch =
  cppBoolSwitch1, ['-'] |
  cppBoolSwitch2 |
  cppEnumSwitch |
  cppStrArgSwitch, stringLiteral |
  cppListStrArgSwitch, stringLiteral |
  cppPchSwitch [stringLiteral];
```

Grammar for C++ compiler switches

A.3 Resource Compiler switches

```
rcBoolSwitch =
  '/x' | '/v';

rcStrArgSwitch =
  '/fo';

rcListStrArgSwitch =
  '/I' | '/d' | '/u';

rcCompilerSwitch =
  rcBoolSwitch |
  rcStrArgSwitch, stringLiteral |
  rcListStrArgSwitch, stringLiteral;
```

Grammar for Resource compiler switches

A.4 Linker switches

```
linkBoolSwitch =
  '/NOLOGO' | '/NOENTRY' | '/IGNOREIDL' |
  '/NOASSEMBLY' | '/DELAY:UNLOAD' | '/RELEASE' |
  '/DEBUG' | '/SWAPRUN:CD' | '/SWAPRUN:NET' |
  '/PROFILE' | '/MAPINFO:EXPORTS' | '/MAP' |
  '/NODEFAULTLIB' | '/DELAYSIGN' |
  '/CLRUNMANAGEDCODECHECK';

machine =
  'X86' | 'IX86' | 'I386' | 'AM33' |
  'ARM' | 'EBC' | 'IA64' | 'M32R' |
  'MIPS' | 'MIPS16' | 'MIPSFPU' |
  'MIPSFPU16' | 'MIPSR41XX' | 'SH3' |
  'SH3DSP' | 'SH4' | 'SH5' | 'THUMB' |
```

```
  'X64';

subsystem =
  'CONSOLE' | 'WINDOWS' |
  'NATIVE' | 'EFI_APPLICATION' |
  'EFI_BOOT_SERVICE_DRIVER' |
  'EFI_ROM' | 'EFI_RUNTIME_DRIVER' |
  'POSIX' | 'WINDOWSCE';

opt =
  'NOREF' | 'REF' | 'NOICF' | 'ICF' |
  'NOWIN98' | 'WIN98';

linkEnumSwitch =
  '/MACHINE:', machine |
  '/SUBSYSTEM:', subsystem |
  '/OPT:', opt |
  '/DRIVER', [':UPONLY' | ':WDM'] |
  '/ASSEMBLYDEBUG', [':DISABLE'] |
  '/INCREMENTAL', [':NO'] |
  '/MANIFEST', [':NO'] |
  '/FIXED', [':NO'] |
  '/TSAWARE', [':NO'] |
  '/LARGEADDRESSAWARE', [':NO'] |
  '/LTCG', [':PGINSTRUMENT' | ':PGOPTIMIZE' |
      ':PGUPDATE'] |
  '/VERBOSE', [':LIB'] |
  '/ALLOWISOLATION:NO' |
  '/CLRTHREADATTRIBUTE:', ('MTA' | 'STA') |
  '/CLRIMAGETYPE:', ('IJW' | 'PURE' | 'SAFE') |
  '/ERRORREPORT:', ('NONE' | 'PROMPT' | 'QUEUE');

linkStrArgSwitch =
  '/DEF' | '/BASE' | '/ENTRY' | '/IDLOUT' |
  '/TLBOUT' | '/TLBID' | '/PGD' | '/ORDER' |
  '/MANIFESTFILE' | '/OUT' | '/VERSION' |
  '/IMPLIB' | '/PDB' | '/PDBSTRIPPED' |
  '/MIDL' | '/KEYFILE' | '/KEYCONTAINER' |
  '/MAP';

linkListStrArgSwitch =
  '/MANIFESTDEPENDENCY' |
  '/ASSEMBLYMODULE' | '/ASSEMBLYRESOURCE' |
  '/ASSEMBLYLINKRESOURCE' |
  '/INCLUDE' | '/DELAYLOAD' |
  '/NODEFAULTLIB' | '/LIBPATH';

reserve = intLiteral;
commit = intLiteral;
libraryName = stringLiteral;

linkSwitch =
  linkBoolSwitch |
  linkEnumSwitch |
  linkStrArgSwitch, ':', stringLiteral |
  linkListStrArgSwitch, ':', stringLiteral |
  ('/HEAP:' | '/STACK'), reserve, [',', commit] |
  libraryName;
```

Grammar for linker switches

A.5 Applying compiler and linker switches

This part of the grammar concerns the incremental addition or removal of compiler or linker switches, either to all configurations or else to specified configurations.

```
cppSpec =
  ('+' | '-'),
  'cpp',
  [ configRefList ],
  '{',
    { cppCompilerSwitch },
  '}';

rcSpec =
  ('+' | '-'),
  'rc',
  [ configRefList ],
  '{',
    { rcCompilerSwitch },
  '}';

linkSpec =
  ('+' | '-'),
  'link',
  [ configRefList ],
  '{',
    { linkSwitch },
  '}';
```

Grammar for applying switches to configs

A.6 Project files

This is the relevant part of the grammar involved with specifying the files that comprise the project. It supports nested directories, wild cards, recursion, flattening, renaming.

```
pattern = stringLiteral;

directoryName = stringLiteral;
folderName = stringLiteral;

(* E.g. +["*.cpp" "*.h"] *)
nameFilter =
  ['-' | '+'],
  pattern | '[', {pattern}, ']';

(* E.g. directories -[] *)
directoriesFilterDef =
  'directories', nameFilter;

(* E.g. +cpp{/O1} *)
fileOpt =
  'exclude' |
  cppSpec |
  rcSpec;

(* E.g. "*.cpp" flat * : +cpp{/D "BLAH"} *)
filesSpec =
  nameFilter,
  [ ['flat'], '*' ],
  [ ':', fileOpt, {',', fileOpt} ];

dirElement =
  filesSpec |
  directoryName, ['as', folderName],
      directoryStructure;

directoryStructure =
  '{', {dirElement}, '}';
```

Grammar for project files

A.7 Entire file

```
varName = identifier;
varValue = stringLiteral;

path = stringLiteral;
logicalPathToProjDir = path;

subProjSpec =
  'subproj',
  '{',
  { logicalPathToProjDir },
  '}';

toolFilesSpec =
  'toolFiles',
  '{',
  { path },
  '}';

description = stringLiteral;
command = stringLiteral;

eventSpec =
```

```
  ('prebuildevent' | 'prelinkevent' |
      'postbuildevent'),
  [ configRefList ],
  '{',
  description,
  command,
  '}';

genKey =
  'OutputDirectory' |
  'IntermediateDirectory' |
  'DeleteExtensionsOnClean' |
  'BuildLogFile' |
  'InheritedPropertySheets' |
  'ConfigurationType' |
  'UseOfMFC' |
  'UseOfATL' |
  'ATLMinimizesCRunTimeLibraryUsage' |
  'CharacterSet' |
  'ManagedExtensions' |
  'WholeProgramOptimization';

genSettingsSpec =
  'general',
  [ configRefList ],
  '{',
  { genKey, '=', stringLiteral },
  '}';

xcpjElement =
  'xcpp' |
  'export' |
  '$', varName, '=', varValue |
  '@import', path |
  cppSpec |
  rcSpec
  LinkSpec
  subProjSpec |
  configsDef |
  directoriesFilterDef |
  toolFilesSpec |
  eventSpec;

xcpjFile =
  '$TARGET_TYPE', '=', targetType,
  '$ROOT_TO_PROJDIR', '=', logicalPathToProjDir,
  { xcpjElement },
  directoryStructure;
```

Grammar for .xcpj file

References

[1] *Xcpp Macro Preprocessor*, D. Barrett-Lennard, Jan 2010, `http://www.cedanet.com.au/ceda/Xcpp%20Macro%20Preprocessor.pdf`.

[2] ISO/IEC 14977:1996(E). *Information technology - Syntactic metalanguage - Extended BNF*, First edition 1996-12-15. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`.