Packages, Assemblies & Nests

Packages

Packages are centric to U++. An executable application is built from a package. A package can also build into a dynamic link library, a static library or a set of object files. A package can be used by other packages. A package corresponds to a single directory with the directory name being the name of the package. The package directory contains the package definition file (a plain text file with a .upp extension), which always has the same name as the package directory. The package definition file contains a list of the source files that make up the package, plus information on what type of package it is, how it should be built and what other packages it uses. The source files for the package are normally located in the package directory and its subdirectories, but they may be in any desired location. A package directory must be located in an assembly nest or in the sub-folders of a nest (see *Package-locations* below).

The package definition file is maintained by TheIDE and you should never need to manipulate it manually. The package definition file is updated at various times by TheIDE, including :

- When you add/remove source files to/from a package
- When you add/remove packages used by the package
- When you change package build settings using the *package-organizer*
- When you change the package build configurations using *main-package-configuration*
- When you change/assign custom build steps for the package

For more detail on configuring packages and assemblies, see this

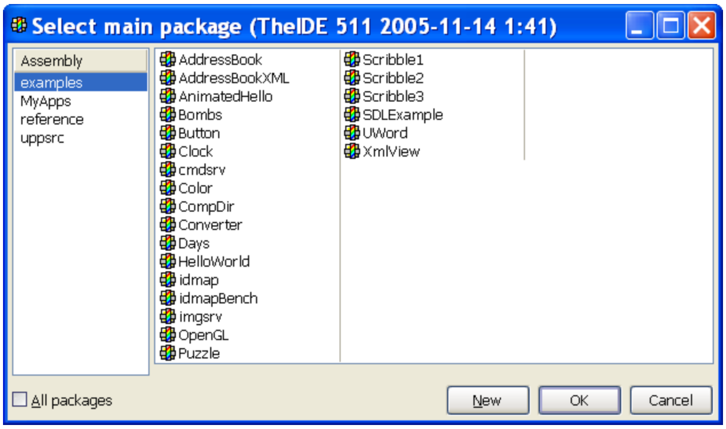
When a package is built, the compiler will be invoked for each of the source files that belong to the package and for any source files that belong to packages directly or indirectly used by the **main package.**

Assemblies

An assembly can be thought of as a collection of packages but it is actually just a set of paths which determine where U++ looks for the packages needed to build a package (or application). The assembly paths also determine where the compiler looks for files named in C++ #include directives. An assembly also specifies the root folder for the location for the output files (.obj, .exe etc) produced by a build plus the location of common files. The paths defined by the assembly are stored in an assembly definition file which has a .var extension and is stored in the U++ root installation directory. A package can be associated with multiple assemblies. Files produced by a build are actually placed in a sub-folder of the output root folder and the sub-folder is named according to the package name, the **build flags** and the compiler that was used. e.g. for the HelloWorld example when built with MSVC++ 7.1 compiler, the output file folder is typically C:/upp/out/HelloWorld/MSC71.Gui.Main where C:/upp/out is specified as the output folder root in the assembly.

Opening a package

To open a package, the *Set-main-package* option on the File menu in TheIDE is used. The "Select main package" dialog will appear.



First select an assembly in the left hand pane, then select one of the assembly's packages in the right hand pane. Click OK and the selected package will be opened in TheIDE with the package name shown in TheIDE application window title bar. The package that has been opened is referred to as the **main package** and it appears first in the list of packages shown at the left-hand-side of TheIDE. To select a different **main package**, the *Set-main-package* option from the File menu can be used. For more detail on creating and configuring packages and assemblies, see this.

In TheIDE, a build command builds a package, not an assembly, but the assembly determines where the needed packages and include files are looked for. The Build option on TheIDE build-menu always builds the **main package**. The build-package option can be used to build any of the packages that are directly or indirectly used by the **main package** i.e. any of the packages displayed in the package list at the left hand side of TheIDE. On the Project menu in TheIDE, the *add-package*, *package-organizer* and *main-package-configuration* options apply to the **main package**. The *project-export* option on the File menu exports the main package and all the packages that are directly and indirectly used by the main package, to any folder you choose.

Nests

U++ requires that packages be organized into nests. A nest is actually just a directory containing a set of package directories and source files. An assembly defines an ordered list of nests (paths) and the packages contained in those nests form the packages of the assembly. The packages associated with an assembly are shown in the right hand pane of the *Select-main-package* dialog when the assembly name is highlighted in the left hand pane. The nest paths specified in an assembly can also be used to set additional include paths for the compiler. See *Include-paths-and-#include-directives* below for more detail.

The assembly containing all of the U++ library packages is uppsrc and the name of the associated nest is also uppsrc.

The U++ examples assembly

The assembly containing all of the U++ examples (such as HelloWorld), is named "examples" and you can see it in the select-main-package dialog. It has two nests, "examples" and "uppsrc". On the Windows platform, the path setting for the nests of the examples assembly might appear as follows :
C:/upp/examples;C:/upp/uppsrc

where C:\upp is the U++ root installation directory. A semicolon separates the path settings for each nest The examples nest contains all of the U++ examples packages and the uppsrc nest contains all of the U++ core library packages.

For portability, **forward slashes should be used in all path specifications** rather than back-slashes. This also applies to #include directives.

Include paths and #include directives

The nest paths defined in an assembly determine where U++ looks for the assembly's packages. These paths are also added to the "include path list" for the compilation of source files via the "-I" (or equivalent)

compiler command line option. e.g. for the examples assembly above, the -I command line setting (for GCC) would be
-IC:/upp/examples -IC:/upp/uppsrc

This means that the path names used in #include directives in C++ source files can begin with the name of a folder/package that is the member of an assembly nest. e.g. in the HelloWorld example hello.cpp file you will see
#include <CtrlLib/CtrlLib.h>

CtrlLib is the name of a U++ library package in the uppsrc nest. Angle brackets should be used in a #include directive when #including U++ source files because this prevents the compiler from looking in the current directory for a folder named e.g. "CtrlLib". i.e. with all U++ supported compilers, the angle brackets means the search for the included file begins with the paths specified in the -I directive. When #including files that are members of the same folder (or sub-folders) as the file doing the #include, then double quotes should be used instead of angle brackets.

On non-Windows platforms, folder names are case sensitive so **it is recommended that the correct case always be used** e.g. CtrlLib and not ctrllib.

See *Alternative-#include-path-mechanisms* below for additional information.

Package locations

A package folder does not have to be located in a top level nest folder. It may be located in a sub-folder of a nest folder if desired. For example, consider the following directory structure.

Nest1/Pkg1
Nest1/Project1/Pkg2
Nest1/Project1/Client/Pkg3
Nest1/Project1/Common

and an assembly nest path setting of
C:/upp/Nest1;C:/upp/uppsrc

Pkg1 is located in the top level Nest1 folder. Pkg2 is located in the Project1 sub-folder of Nest1. U++ searches all of the sub-folders of a nest to maximum depth when looking for packages.

A source file in Pkg3 can #include a source file in Pkg2 (File2.h) with
#include <Project1/Pkg2/File2.h>
Double quotes can be used instead of angle brackets providing the Pkg3 folder does not contain a Project1 folder.

A source file in Pkg3 can #include a file from its own folder (File3.h) with either
#include "File3.h"
or
#include <Project1/Client/Pkg3/File3.h>

The organization of packages and #includes shown above allows the Project1 folder to be located in any nest.without changing any of the #includes.because all of the pathnames begin with Project1. It also allows an assembly to switch between different revisions of Project1 just by changing the assembly nest path setting.

When a package is created using the Create-new-package dialog, the package name must include some path information if the package is not a top level folder within a nest. e.g. for the Project1/Pkg2 package above, the package name needs to be entered as Project1/Pkg2. For Pkg1, the package name can be entered as just Pkg1 because the package is in a top level nest folder.

An assembly may contain multiple projects/applications or just one project. If there are multiple projects in an assembly then you need to consider the package/folder/file name issue described below.

Package/folder/file names

It is usually necessary that the names of the folders and files that appear in the top level nest folders of an assembly be **unique across all the top level nest folders of that assembly** unless duplicated names are referenced using a complete path specification. This means that if an assembly includes the uppsrc nest, then the other nests of that assembly must not contain packages or folders that have the same name as folders/packages in the uppsrc nest. e.g. The HelloWorld examples nest cannot contain packages with names such as CtrlLib, RichEdit or Common because these are the names of uppsrc library packages.

Hence the names of uppsrc packages need to be avoided when choosing names for folders/packages that are located in a top level nest folder if the assembly includes the uppsrc nest. Refer to the uppsrc folder in the U++ installation path for the full list of uppsrc package/folder names. The names of uppsrc packages need to be avoided as folder names within any folder that is specified as an include path to the compiler (see *alternative-include-path-mechanisms* below).

If a package is to be distributed to others, one way of avoiding a clash of package names is to locate them in a folder whose name is likely to be unique e.g.
Nest1/CZ1Soft/Pkg1.
and
#include <CZ1Soft/Pkg1/File1.h>
CZ1Soft is a name that has a reasonable chance of being unique. The name of the Pkg1 folder can be anything because it is not a top level nest folder and.is not specified as an include path to the compiler.

If source files are placed directly in a nest folder (e.g. Nest1 above), then the names need to be unique across all nest folders of the assembly unless they are always accessed with either a complete path specification or with no path specification (in which case they need to be in the same folder as the including file). e.g.
#include "File1.h" contains no path specification.
#include "../Pkg2/File2.h" is a complete path specification.
#include <Project1/Pkg2/File2.h> is an incomplete path specification.

Note ".." in a path specification means "parent folder" i.e. up one level

Alternative #include path mechanisms

The nest paths specified in an assembly are normally used to identify the directory (or directory tree) where packages can be found and, as explained above, the "normal" method of #including header files is that when the header file name involves an incomplete path, the given pathname begins with the name of a top-level nest folder e.g. #include <CtrlLib/CtrlLib.h>. CtrlLib is the name of a top-level nest folder because it is located in the uppsrc nest.

Because the paths specified in an assembly are added to the "include path list" for the compiler (using -I or /I), you can use this mechanism to add directories to the include path list, even if those directories don't contain any packages. You need to remember that the search for packages looks in all sub-folders of the assembly nest paths and also that you may need to avoid using folder names that are the names of uppsrc packages (see the *Package-folder-names* section above). The order of the -I (or /I) directives supplied to the compiler is the same as the order of the nest paths specified in the assembly and this determines the search order when the compiler looks for #include files.
e.g. suppose you have a folder, C:/SomeFolder, that contains a header file SomeHeader.h. You can add SomeFolder to the assembly nest path like this
C:/upp/examples;C:/upp/uppsrc;C:/SomeFolder

In your source files you can now write
#include <SomeHeader.h>
or
#include "SomeHeader.h"
Angle brackets are preferred because they mean the search begins in paths specified in -I directives rather than in the folder containing the file doing the #include
.
Providing the header files in the SomeFolder folder use angle brackets when #including uppsrc files, it will not matter if the SomeFolder folder contains directories that have the same name as uppsrc directories, because C:/SomeFolder is last in the assembly nest path list.

You can also use the *package-organizer* in TheIDE to specify additional include paths. The *package-organizer* allows you to enter additional switches to be passed to the compiler for all packages, for specific packages or for specific source files. This allows you to add a -I switch (or /I) to specify an include path. These switches get added to the compiler command line *after* the -I switches for the assembly nest paths. To see how the compiler is invoked when a source file is compiled, turn on the *verbose* option in the Setup menu in TheIDE. You can use **build flags** to restrict the additional compiler switches to be in effect only when your own package's source files are being compiled. The *package-organizer* also allows you to specify compiler switches for specific files.

Layout file #includes

A layout file contains a description of the GUI part of a project i.e. the layout of widgets etc. e.g. the AddressBook example in the examples assembly uses a layout file and the AddressBook.cpp file has the following two lines.

```
#define LAYOUTFILE <AddressBook/AddressBook.lay>
#include <CtrlCore/lay.h>
```

The #define for LAYOUTFILE should use angle brackets and not double quotes and must also include a path specification that starts with a top level folder of a nest. i.e. it should not be written as
`#define LAYOUTFILE "AddressBook.lay"`
because the CtrlCore/lay.h file uses it to #include the layout file (multiple times) with
`#include LAYOUTFILE`

<PackagesAssembliesAndNests$en-us - last updated: Jan 3rd 2006>