Application Wizard

Description

This is a piece of software that can be used to write other pieces of software! It can provide template code to make the whole development process that much less painful.

This wizard is similar to the Application Wizard found within Visual Studio but it is an external application that could be used to build Linux projects, or ApplCore projects.

What the Wizard does

What the wizard does is takes a number of template files and wraps them up into a single wizard DLL, which can be easily distributed. When the user enters a project name and a file path in the wizard and hits "Go" then the wizard extracts some or all of these files, optionally replaces text in the files (to name them the same as the project name or to include other options). The wizard can then be used to build project files allowing C++ projects to be imported into Visual Studio.

Example Wizard Uses

- Building plugins for the Wizard (!!)
- Building Qt custom widget applications
- Building C++ BNCS Driver / Panel applications
- Building ApplCore panels

Architecture

The Wizard uses a system of DLL "plugins". This means that there is the basic shell of the wizard and other modules can be added (hopefully by a number of authors) as and when required.

Wizards are authored in Qt, but don't necessarily have to build Qt applications.

The Wizard shell itself does very little – you select the project type you want to build, and then the location of where you want to put the template code. The shell then just calls the plugin.

The plugins do all the actual work. Typically the plugins will have all the template code embedded into the DLL.

Simple plugins will do no more than just extract the template code and then perhaps build a Visual Studio project file.

Complex plugins may offer a number of different options and then produce a project with only the relevant files / sections included.

Building makefiles or Visual Studio Project Files (.dsp) for C/C++ projects

There are two methods of doing this – the easy way or the hard way!

The easy way:

- take a working example of your finished application (of the sort that your Wizard is to produce). Then simply replace the relevant bits within the make/dsp file.

The hard way:

- build a <project>*.pro* file and use *qmake.exe* to translate this into the relevant project files.

I would strongly recommend the first approach (thankfully the easy way).

The "hard way" also requires that the end user of your wizard has to have Qt installed on their machine.

Building make/dsp files by text replacement

Take your working make/dsp file and look for all instances of the files that you need to change the name of – generally this will be files the same name as the project name that the user enters into the Wizard.

Replace the filenames with the text replacement strings %%NAME%%.

Treat the make/dsp file in the same way that you would a normal wizard project file – doing the normal text replacements as the file is extracted.

Building make/dsp files using QMake

I'm not going to explain this here – this requires more intimate knowledge of QMake than I'm prepared to document!

Embedding and Extracting Template Source Files

The Qt utility *qembed.exe* enables embedding any file into a normal header file suitable for inclusion into a Qt project. It is a command line tool.

e.g.

```
qembed main.cpp main.h > embedded.h
```

this embeds the files main.cpp and main.h into the file embedded.h.

qembed.exe is not normally built as part of the normal Qt install and you have to do this by hand – just open "*C:\Qt\tools\qembed*" (assuming you installed Qt in "*c:\qt*") and build qembed.exe from the project file (qembed.dsp). Move the resulting executable into "*c:\qt\bin*".

Part of the embedding process also includes the code to extract the data. If you open your header file you'll find:

```
static const QByteArray& qembed_findData( const char* name )
```

Using the embedded files

There are some BNCS helper functions to extract this file and save it:

e.g.

```
void wiz::savefile( QString in, QString path, QString replace = "" );
```

This function takes an input file (contained in the string "in") and saves it to the path "path" and optionally replaces %%NAME%% placeholders with the "replace" string given.

e.g. the following extracts the file "main.cpp" from the data embedded in the header file, saves it as "main.cpp" in the destination path handed to the plugin by the wizard and replaces any instances of %%NAME%% with the string "stuff".

```
wiz::savefile( qembed_findData( "main.cpp" ), path + "main.cpp",  "stuff" );
```

The second of these functions can do multiple replacements within the input string by passing it a list of tag/text pairs:

```
void wiz::savefile( QString in, QString path, QStringList replacelist );
```

e.g. Consider the following source file:

```
void main( void )
{
	%%PLACEHOLDER1%%
	%%PLACEHOLDER2%%
}
```

we want to replace the place holders with data. In the following example we'll replace PLACEHOLDER1 with a simple function call and PLACEHOLDER2 with the entire contents of another embedded file.

```
QStringList sl;

sl	<< "%%PLACEHOLDER1%%" << "printf( \"hello world\" );";
sl	<< "%%PLACEHOLDER2%%" << qembed_findData( "bit.cpp" );

wiz::savefile( qembed_findData( "in.cpp" ), fullpath +"in.cpp", sl );
```

In this example the “%%PLACEHOLDER1%%” is replaced with a simple hello world function. “%%PLACEHOLDER2%%” is replaced with the whole of another embedded file called “bit.cpp” (note the compiler takes care of the cast required to convert the QByteArray returned by the de-embed function to a QString).

You may want to optionally replace the placeholder depending upon the options that the user selected for this build in which case you’d see something like:

```
if( sayhello_option_selected )
      sl    << "%%PLACEHOLDER1%%" << "printf( \"hello world\" );";
else
      sl    << "%%PLACEHOLDER1%%" << "";
```

the important things to note are that the strings are in pairs, and that you should replace all placeholders with something even if it’s nothing.

There’s one more save function for binary files (executables, data, image etc.)

```
void wiz::savefilebinary( QByteArray in, QString path );
```

e.g.

```
wiz::savefilebinary( qembed_findData( "applcore.exe" ),  path + “name” + ".exe" ));
```

A step by step guide to creating a new Wizard

This wizard will build a new project that builds a very simple Windows console application that prints “Hello” and the application name. This means you have to do all the steps required to build any C/C++ Visual Studio Wizard.

Run the Wizard and select the “Create new Wizard” option, and enter a name and path. Lets assume to help in this explanation you’ve called the new Wizard “mywiz”.

Hit OK to create the new template Wizard.

A dialog should then confirm correct creation of the makefile and Visual Studio project file.

Navigate to wherever you created the new Wizard files and open the “mywiz.dsp” file with Visual Studio.

You should then have a project with all the necessary files to build a new Wizard.

Lets run through the files:

mywiz.cpp / mywiz.h	are the files you will have to do most work with to create your wizard project. The files are created with a large block of comments that gives some sample function calls.
wiz_util.cpp / wiz_util.h	contain useful utility functions (like file

	saving functions) that should make writing your wizard as easy as possible. You shouldn't have to modify these but they are there if you want them.
_readme.txt	a short readme file to explain some of what you have to do to embed files.
bncs_wizard_plugin.h	This contains the base class of your wizard function and the code necessary to make it a plugin – you don't need to do anything with this – it just needs to be included in mywiz.cpp
embedded.h	This is where all your embedded code/files need to go. See above about using qembed.exe to embed your files into this header

We need to create the file that will be used as the template for the application we're going to build.

We'll call this main.cpp and it should look something like this:

main.cpp:

```
#include <stdio.h>

void main( void )
{
      printf( "Hello %%NAME%%" );
}
```

Notice the fact that we're included a placeholder `%%NAME%%` for the application name in the code.

We need to do the following steps for this or any other file we're going to include in a Wizard.

- Embed the file into the source code – do this by right clicking "embedded.h" in your project view and hit settings. In "Custom Build"/"commands" type something like: c:\qt\bin\qembed.exe main.cpp > embeded.h
- Put a line to extract this file in the "go" function in "mywiz.cpp". This is going to look like:
  wiz::savefile(qembed_findData("main.cpp"), fullpath + "main.cpp", name);
  What this does is extract the file main.cpp, save it to the path provided to us (which is

what the user typed in). In this case whilst saving it, replace any instances of %%NAME%% with the application name.

That gets the source code saved into the right directory.

The tricky bit

It's not anything to do with this wizard as such:

Work out how you're going to structure your project, what files you're going to need and what options it requires. Are you just going to extract a few files and rename them to be your project title? Are you going to extract lots of files, optionally inserting components and files depending upon what the user selects as a configuration option, then having done all of this build a Visual Studio project file (.dsp)?

I recon working out how to structure your components is far harder than trying to work out how to include any of this in the wizard.

Command Line

Version 4.5.1.0 added the ability to use command line parameters so that this tool may be run to automatically generate code (for example from the Visual Editor).

Param	**Example**	**Use**
/path	/path=c:\blah	The path to use for this new project
/project	/project=proj	The directory to create in that path
/key	/key=Create C++ Scripted UI / Panel	The plugin to use to create this new project. **Note:** this has to be exactly the same as it appears in the listbox of the

There can be as many other command line options as the plugins themselves define. For example the "Create C++ Scripted UI / Panel" wizard (bncs_script_wizard.dll) defines the following additional parameters

Param	**Example**	**Use**
/panel	/panel=p1	The name of the panel to create. This parameter is optional and if not supplied with use (and create) "p1"
/title	/title=This is the title	The title to use for this dialog. This parameter is optional, and a dialog is shown to enter this information if this is not provided