Subject: Re: Building U++ for MinGW32
Posted by mirek on Thu, 10 Jul 2014 07:48:30 GMT
View Forum Message <> Reply to Message

pcfreak wrote on Mon, 07 July 2014 05:53Hello,
thank you for the reply. I wanted to build U++ as library with my MinGW so that I can integrate it into my tool chain.
Too bad to hear that this is unsupported. Are there plans to support this?

Now, the question is what you really mean by toolchain.

One of defining features of U++ is the transparent modularity system (assemblies, nests, packages), which needs its own build system - that is why theide exists. It is possible to use commandline version of this build system ("umk"). It is also certainly possible to forget about it and try to create 'normal libraries'.

There is one not really resolved issue with converting U++ into normal library. U++ is using "global constructor trick" to initialize its parts.

To explain by example: there are various raster graphics formats supported by U++, and the support for each format has its own module ("package", e.g. plugin/png). If your application needs to support jpeg, you add "plugin/jpg" into the project. Now there exists in U++ Image basic libraries function that is able to load image in any format. Obviously this function needs to have a list "format loaders", which means that "plugin/jpeg" has to register itself so that this function knows about it. The act of registering is done using global constructor magic, which in reality looks like (with the use of macro)

```
INITBLOCK {
 StreamRaster::Register<JPGRaster>();
}
```

- INITBLOCK is the code, that has to be run on the start of application, before 'main'.

So far so good. But if you put this as single thing into say "init_jpeg.cpp" file, create a library from several files and link with your application, linker will simply exclude "init_jpeg.o" because there are no references to its code from the application.

That is why U++ build system (besides of course many other things) recognizes ".icpp" extension, which means ".cpp code that always has to be included in final product". (Of course, U++ build system does much more than that too :)

Now, any effort to change U++ into library has to deal with this in some way. One possibility is to change the U++ library code, replace all INITBLOCKs with some sort "InitXXX" function and make it mandatory for client code to call them all. Perhaps not very user friendly, as there are now total 109 INITBLOCKs in base U++ now...

Alternative solution, implemented a couple of years ago, is a little addition to theide that

automatically creates "init" files that look like this:

```
#ifndef _plugin_jpg_icpp_init_stub
#define _plugin_jpg_icpp_init_stub
#include "Draw/init"
#define BLITZ_INDEX__ F227b3ee1134af92a2493f791a66e2afe
#include "jpgreg.icpp"
#undef BLITZ_INDEX__
#endif
```

- they include all .icpp files in package and also init files of packages used by package.

The idea here is that your application would #include just top-level <init> files in its main .cpp, which would provide a safe way to call all INITBLOCKs.

I guess this is perhaps smart, but to my knowledge, nobody used it yet in practice.

Other than .icpp issue, I would say U++ as library might have the issue with extreme modularity it has. I am afraid that it will be a bit hard to decide which packages to group into what libraries. But I guess it is doable.
