ODE Manual Compilation

If you’re unlucky enough to be compiling the ODE source code on a platform where no previous version of the ODE tools will run, here’s a brief guide to building the source code by hand. Or at least enough of it so that you can build it with itself.

The only tools you’ll need to compile and link are the following: mk, workon (or build), and mkbb. Each of these will of course require the entire src/lib tree to be built (and a static or dynamic version of that library code created).

I usually create a shell script (if the platform supports such a thing) to make things easier, just a one-liner that includes all of the correct –D (etc.) flags for the compiler. These flags can be determined by looking at our existing suite of platform-specific makefiles (e.g., x86_nt_4.mk) under src/rules_mk. You’ll have to create one of these files to build with ODE anyway, so look in a sample makefile that’s closest to the platform you’re building on. The makefile variables you’ll need to look at for the compile phase are CDEFS, CFLAGS, C_CFLAGS, and/or CPP_CFLAGS. Take a look at x86_nt_4.mk and ia64_hposs_6.mk no matter what you do, so you can see some of the ODE-specific definitions that may be expected (e.g., NO_BINARY_OPENMODE, BOOLEAN_AS_MACRO, etc.). ODE has been ported to so many machines that there usually is some macro in the source code that controls machine- and compiler-specific behavior. You shouldn’t have to change any source code, rather just find the right macros to turn on or off.

There is also one additional –D flag that is set from src/Makeconf, that being the ODEDLLPORT. This variable always needs to be passed during compilation, but only has a real value for Windows. So your compilation on all other platforms would just look like this:

```
g++ –c –DODEDLLPORT= -DUNIX –DETC foo.cpp
```

If this is a completely new platform that has never been supported in a previous version, you’re going to have to make some code changes as well, mainly (hopefully ONLY) in src/lib/portable and src/lib/portable/native. The main thing is adding a new platform definition in lib/portable/platcon.cpp. If you take a look at this file, you'll see that it's pretty easy to add a new machine name. You can control which macro conditional will be true by setting vars like –DSOLARIS and –DSOLARIS_SPARC in your platform-specific makefile (and shell script, for now). If there already are definitions of this sort, chances are you need no change. If you're simply supporting a newer version or a different compiler build, you can probably get away with tweaking settings in the platform-specific makefile (i.e., changing CCFAMILY or CCTYPE).

Caution should also be made if you think you need to change any makefiles in rules_mk (such as std.mk). The common makefiles are so variable-driven that most functionality can be altered by simply changing/setting the right variable(s). This requires some research and experience with ODE, but you should always ask yourself if you can cause a change to the compiler/linker command name or arguments by simply tweaking a variable. Most often, you can. If you're adding support for a completely different/new compiler, though, that would certainly warrant altering std.mk.

A shell script to call your compiler with the correct –D flags (etc.) might look something like this:

```
#!/bin/ksh
g++ –c –DODEDLLPORT= -DUNIX -DNO_BINARY_OPENMODE $*
```

Then you're ready to build the library. Change into the src/lib/portable/native directory and try to compile all of the files listed in Makefile as OBJECTS (note PORTABLE_NATIVE_OFILES is defined in both Makeconf and in platform-specific makefiles, and is mainly used to choose the correct *arch.c to compile. It also includes the gregex.c file on Windows platforms (a frontend for a 3rd Party regex package, since MSVC doesn't include a regex package), as well as cutime.c on OpenVMS (a 3rdParty utime function, as VMS doesn't have one in its default library). You may need to add new macro defines (-D flags) to your shell script if a file doesn't compile right. You should be a programmer if you're reading this, and thus should be able to figure out what function call(s) are appropriate for given situations. The existing defines in the code for the plethora of other platforms should make it easy. Generally speaking, you shouldn't need to make code changes, just changes to the –D list.

Then, cd back up to lib/portable and build the .cpp files there (all of them except dll.cpp, which is only for Windows). This dir contains a file "instant.cpp" which may cause you problems, since it contains an explicit list of template instantiations. Many of the tools under src/bin contain these as well. You'll need to determine whether your compiler requires these to be defined with #pragma, or with normal C++ definitions (AIX, OS/400, and Windows are known platforms where the default compiler requires #pragma, so in the platform-specific makefiles for these you should find –DUSE_PRAGMA_FOR_TEMPINST defined in CDEFS).

Then you can build the remaining subdirs of src/lib separately, all of which should compile ok if the two portable directories went ok.

Once you have all the object files created, you can create a static library containing all of them. This is usually as simple as going up to src/lib and doing "ar cr ode0500.a io/*.o portable/*.o …".

Now to build the tools you'll need.

Go to src/bin/make and compile all the .cpp files. Then link it with something like "ld –o mk *.o ../../lib/ode0500.a". Try running it to print out the usage text: "mk -?". If that worked, pat yourself on the back…you're almost ready to rebuild the source code with ODE doing it all for you.

Then cd to ../workon and do the same thing (with "-o workon", obviously).

The mkbb command is a bit more complicated, as it requires objects from both the mksb directory and the mklinks directory. First, go to bin/mklinks and compile mklinksc.cpp. Then go to bin/mksb and compile mksbc.cpp. Then go to bin/mkbb and compile all of the .cpp files. Link with something like "ld –o mkbb *.o ../mksb/mksbc.o ../mklinks/mklinksc.o ../../lib/ode0500.a".

Then copy these three commands to somewhere in your PATH (e.g., $HOME/bin), make sure you have at least "rx" permissions set on them.

Now then, you'll need to put all those –D flags you used (once you know what they all are) into the appropriate platform-specific makefile. As hinted at earlier, I usually copy an existing similar makefile to the new platform convention (arch_os_ver.mk, so e.g. x86_solaris_10.mk). Or I just include a preexisting one and add/change things in that one as needed (look at x86_nt_5.mk).

You'll need to erase all those object files, the library, and the executables you created before (otherwise ODE will find them when it builds and use them instead of recompiling from scratch…you need to make sure it will build with itself, after all).

Then you'll need to run mkbb to create a backing build structure…you can do this somewhere above the existing src directory and then just move the src dir into the backing build. Or if the parent dir of "src" is already a suitable backing build name, go to the parent of *that* dir and use that dir name as the bb name. So if you have the source code here:

```
$HOME/ode/src/…
```

You can do this:

```
cd $HOME
mkbb –dir . ode
```

Then you're ready to run workon (should work with no args). That will put you into the src directory of the backing build. Then, for this run, define NO_TOOLSBASE in your environment so it'll just use the ODE tools from your PATH. Then just type "mk" and let it rip. You may get compiler errors, but figuring those out are outside the scope of this document.

Good luck!