What Makes a Good Software Build System

Reginald H. Beardsley

A good software build system allows describing any arbitrary software structure and any arbitrary source transformation process with perfect separation of what is to be done from how it is to be done. Above all, it must not overwrite the files used to control the build process. The build process itself should have only minimal dependencies.

Simplicity is the predominate requirement. This mandates that all the the system specific details of the transformation process be isolated in a single file. Ideally the person building the software should not need any knowledge of the structure of the software. However, optional software dependencies often make that goal unachievable, so the best one can mandate is minimal knowledge of the software package to be built.

Discoverability is the other important requirement. This mandates that the system configuration file have a clear explanation of what each symbol used in the transformation process does. While there is a long history of symbol conventions associated with make(1), their usage has been muddied by the widespread use of atutoconf, automake and libtool such that things for which the definition was obvious are no longer clear. Some of the ambiguity results from the introduction of dynamic linking and shared libraries which were not common when the make program was created. The rest is the result of machine generation of the control files by software written by programmers with limited knowledge of the target system.

With large software systems, the default console stream produced by the build is often too verbose to be convenient for monitoring the progress of a build. However, the all too common practice of turning off the default console output and substituting abbreviated progress messages is not the way to do this. The proper method is to pipe the default output to tee(1) and write it to a log file that can be checked with automated tools that scan for errors and warnings. A subsequent process then filters its input and presents only the high level detail to the screen.

All errors and warnings should be corrected in the source if possible. Every time the software is built, the log file should be checked for errors and warnings that compares the output with previous builds. There are, however, always cases where certain systems complain about constructs that in fact are valid. The program that checks the detailed build output should screen for and suppress known messages so that any new messages are readily apparent.

The remaining requirements for a good build system are isolating object files by system and automated implementation of unit level regression testing. There are many ways to implement the former. Having a separate file tree for all the objects is popular. However, this has the defect of requiring explicit naming of the output path for objects and often gets excessively verbose. The main alternative is to build in a system subdirectory and reference the source files in the parent directory. This is particularly handy if the build system also doubles as the development system as it makes clear what target systems have been built. It works particularly well if the same directory holds the area for running the regression tests.

Regression testing is all too often neglected by developers. Frequently testing is performed by another group. While some degree of post integration testing is required, fairly simple unit level tests will do a far better job of ensuring low defect levels in released code. If a simple driver is included in the source file and enabled by conditional compilation, it is particularly easy to automate the testing process. For mathematical routines, a particularly elegant approach is to test for an environment variable that matches the function name. If the environment variable is defined, the module then dumps all the input to an ASCII file in the format read by the test driver. This

then produces a test file which can be incorporated in the regression test suite to verify that a bug is fixed. It also allows a user to set an environment variable, repeat an operation by way of a GUI and get a reproducible bug report without any extra effort or knowledge.

While make is more elaborate than needed for a one time software build, its widespread use and familiarity justify using it as the basis for a build system. This also facilitates software development and debugging. Though the system versions of make are frquently problematic, Gnu make is readily portable, robust and has sufficient features to accommodate any requirement.

In summary, Keep It Simple Stupid applies to building software as it does to everything else. For large systems it can be difficult to achieve, but the results over the long term justify the effort demanded. In any event, the notion that anyone can write software that reliably determines how to process source code on any arbitrary target system is naive at best.