Using ROOT, Windows, and Linux at DØ in Physics Analysis

Gordon Watts
University of Washington

We Was Robbed!

CHEP 415

Why Go Against The Flow?

You can tell what phase an experiment is in by looking at the ratio of Linux machines to Windows machines. Windows machines dominate during construction — most likely because of their superior productivity tools (Excel, Project, etc.) and Linux during analysis — most likely because of their superior price/performance and native command line environment. In reality both platforms have something to offer for both phases. I have always thought the build/edit tools and debugger on Windows were better than their counter parts on Linux. This poster describes the work necessary to take advantage of the build and debugging tools on Windows in a Linux analysis environment.

The DØ code and build system was not designed with Windows in mind. Porting can be loosely be divided into two tasks:

The Raw C++ code. The algorithms, the plotting packages, access to the data storage layer.

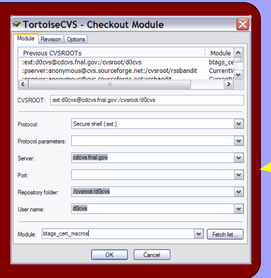
External Access. This includes all interfaces to the operating system, file system, command-line environment. In particular, the build system, the data storage system, etc.

As it turns out, modern day compilers make Task 1 very easy. Only the most complex template code causes difficulty. Task 2, however, is quite difficult, In particular the data storage and the build system. Fortunately, DØ uses ROOT to store its high level analysis objects, and that is already cross platform. Particle physics build systems are designed for a production environment. They are very sophisticated, often based on the infinitely flexible Unix **make** tool. It is almost impossible to write a general translation tool between a make file and the build tool as is contained in Microsoft's Visual Studio. The build system translation was the most difficult part of this project.

This is not meant to be a 100% translation of the Linux environment. This only works with executables that have limited dependencies. Full DØ reconstruction, for example, would be much more difficult.

Windows CVS Client

There are several windows CVS clients available. The local favorite is an open source version, called TortoiseCVS (hosted on SourceForge.net). It is completely integrated with the Windows Explorer GUI and capable of doing everything required for DØ software development except the *rtag* command.



Kerberized SSH

We've used an old version of cygwin's ssh and added Kerberos to it. Modern versions could probably be used out of the box.

DØ CVS Repository

The CVS repository is the link between the Linux and Windows worlds. Almost all interaction between the two worlds takes place through this repository. For DØ the repository is maintained by the Computing Division on a machine accessible only through kerberized ssh.

Windows *Linux*

Linux CVS

At DØ this is a combination of using the CVS command directly and a few DØ specific helper scripts (to keep symbolic links correct).

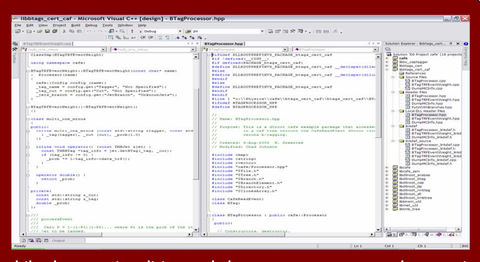
Window File System

The directory structure for source code extracted from CVS is identical on both platforms. Only location of the built libraries and binaries is different.

Linux File System

CVS packages (source code) and built binaries are laid out in separate directories. Source code contains build instructions. The source code layout is identical on all platforms.

Integrated Development Environment



The IDE is an editor, a debugger, and a build system integrated in a single application. For example, the editor can take advantage of the compiler and use object and symbol information from the compiler to present the user with a list of methods relevant for a variable name as they type. The IDE will run the compiler in the background while the user is editing and show syntax errors as the user is typing. One can modify source code directly in the debugger, or type symbolic expressions directly into the debugger that use the language's syntax. The build system makes it possible to quickly compile a single file which dramatically speeds up the development process. The IDE is designed to reduce the write-compile-run cycle. Stand alone tools are better for individual tasks (for example, EMACS is a better editor), the IDE, with its tools bundled together, is more compelling.

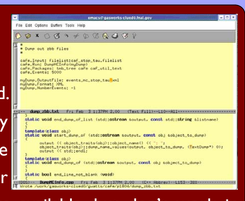
DØ Specific Extensions to the IDE

The IDE specifies its projects (build instructions) using XML, and DØ specifies its using CTBUILD. The IDE is extensible, however, and so a number of plug-ins were added to aid integration:

1. **Create a DØ Build Project.** This is similar to setting up the DØ build environment (a local release). It will scan a directory for all checked out cvs packages and convert them to IDE libraries. Rootcint files are supported and sharable libraries are supported as well.
2. **Automatic Fetch.** If cvs packages are missing during a built the system can search the DØ cvs repository for them and check them out and build them automatically..
3. **GUI to add new Components.** This takes care of generating the 4 files required to add a new object to a cvs package as well as updating the CTBUILD directives.

The work to build this code took the most effort. And it is the place of most active continuous development.

Editing



EMACS or teco are used. The interaction is strictly with the file system; there is no direct compiling or debugging (extensions are available, but don't work in HEP environment with out work).

Command Line

All action happens from the command line in Linux: editors are started, the debuggers run, the build system invoked, and jobs are submitted to the batch farms. Powerful script systems available from the Linux command line make it simple to code complex job control (and build systems). Windows has no counter part to this yet. However there is hardly any integration between the editor, debugger, and build system, once the individual programs are started.

Debugging

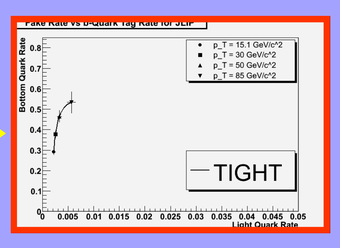
Most programmers use gdb or a x-Windows compatible debugger. The GUI tools tend to have buttons mapped directly to gdb commands rather than being designed to be a GUI debugger from the start. These debuggers are powerful, but only for experienced users master their power (see text box below).

Batch Farms

On Windows one may test on 10,000 events, but on Linux one frequently runs on millions of events using large farms or the GRID. After running, job data is recombined for final results. Modern analysis jobs require many millions of events and quick access.

Executable

The IDE build produces a standard executable. A ROOT data file, used as input, is usually copied over from Linux using SCP. Cross platform scripting is used to massage final plots (pyROOT, for example). One typically runs on ~10,000 events for testing.



The Final Result

Why Is Using A Debugger So Hard?

Printf vs. GDB

Efficient debugging in a particle physics environment is difficult. Most physicists use the old *printf* style of debugging – even ones with advanced programming skills. A debugging tool allows you to follow code execution paths, examine arbitrary variables (not just the ones you chose to print out), and even pause the program for further examination when a unique condition arises.

Difficulty using GDB

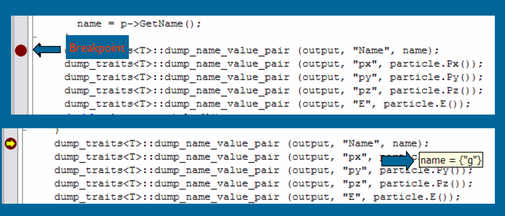
There is a learning curve associated with a debugging tool whereas there is almost no learning curve associated with *printf* statements. As a result, debugging tools are used by depressingly few physicist.

Anecdotally, I hear two main reasons for this:

1. The gdb tool is too complex to use, and
2. Finding their code is difficult.

I suspect the second strongly influences the first. Modern day physics analysis frameworks are very complex. A physicist writes only a small piece of plug-in code. Modern frameworks load this plug in code as a shared library. Setting a break point on the first line of code is a chore: when a tool like *gdb* starts up it hasn't even loaded the physicist's plug-in module and so can't set a break point! There are ways around this, and gdb's powerful macro features can be used to simplify the process. But none of us spends the time to do this! And even once you get to your break point, examining variables in *gdb* is not intuitive. I should note that there are GUI front-ends for gdb, but I've never stuck with them because they don't support enough features or are more difficult to use than the command line interface. TotalView is the best Linux GUI debugger I've used.

The IDE Debugger



One of the big advantages of an Integrated Development Environment (IDE) is that source code editing, debugging, and building all occur from the same interface. Setting a break point in your code is as simple as point-and-click, and then hitting the *run* button. Variable values can be determined with a simple mouse-over. Under some circumstances you can even edit your code while running and have it re-built mid-run. These features make it easy for a beginner (or a physicist who doesn't use gdb often enough to remember the commands) to use a debugger.

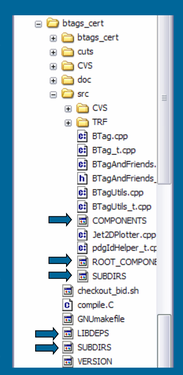
Simplifying the Build System: CTBUILD

Complex Build Systems Are Natural Outgrowths of the Tools

The DØ build environment is based on Software Release Tools (SRT), a collection of makefile fragments. Each source package contains makefile fragments that direct SRT's build process. The implications of allowing 100's of physicists to modify makefile fragments are serious. For example one makefile fragment can affect another through global variable declarations. Or because raw makefile code is exposed it can be very difficult to modify the build infrastructure because a makefile fragment depends on an undocumented feature.

Simplified Build System

Most analysis packages require a simple set of rules: a list of source files, the location of includes files and perhaps a few well defined preprocessing steps (e.g. *rootcint*).



Dave Adams, a former member of DØ, developed CTBUILD to address these concerns. Think of CTBUILD as an abstract interface to the DØ build system. Almost all DØ packages now use CTBUILD. CTBUILD is controlled by a number of special directive files. For example, the COMPONENTS file contains a list of the source files to build; the ROOT_COMPONENTS file lists the files that should be processed by *rootcint*. No explicit commands are specified in the CTBUILD directives.

The fact there are so few options has many advantages. The release build managers, responsible for building all of DØ software (100's of packages), have a much easier job; they can modify SRT and as long as they honor the CTBUILD interface all CTBUILD packages will work. No so with packages based on make file fragments. The restricted CTBUILD interface also lends itself to translation to other build systems besides SRT. CTBUILD maps well onto the MSVC project and solution model. The work described here supports only CTBUILD DØ software; other software must be converted by hand.

Supporting Shared Libraries

Shared libraries, or .so's on Linux and .dll's on Windows, pose a special set of problems because the model is very different on the two platforms. ROOT and the latest incarnation of DØ's analysis framework depend heavily on them. On Linux, every single global symbol is exported in a .so by default. On Windows only specified symbols are exported. ROOT already contains code that will scan an object file for every global symbol and generate an export directive. Global variables, however, are worse. A global variable is usually just an address. But when a global variable is stored in a shared library it is a pointer to an address. In short, the code that accesses the global variable must be altered to add a dereference. This is usually accomplished by marking all imported global variables in header files with a special directive. The header files are parsed and re-generated with the import directives. These modified files are then used to compile the source code. Though the concept is simple, a significant amount of behind the scenes work must occur to make this transparent to the end user.

HEP Software Is Too Complex

How Complex Software Gets Written

The current generation of Linux build tools (e.g. gmake and cmt) are incredibly flexible. These systems are designed for the expert who can accomplish almost any complex build task using these tools. Putting this power in the hands of 100's of non-expert developers – physicists – is an entirely different matter.

No one starts off to build a tool with a complex user interface. The tools we use in particle physics tend to follow similar path. Experts put together the first version of program. This first version was never intended as much more than a temporary stop-gap solution, but is so compelling it grows. As it grows and new options are added, those options are given the same ease-of-use weight as the original vision for the tool. The result is (usually) a command line interface that takes nothing for granted, must be completely specified, and has 100's of options. A new user approaching the system will be lost.

Most of the attendees at CHEP are experts in one form or another. We write much of the software that is used to analyze the physics data. In most cases, however, we don't actually do the physics. Post-docs and students – with much less computing experience – write the bulk of the physics code. Watching them start out in a framework as complex as ATLAS or DØ is a harsh lesson.

We Need More Tools like CTBUILD

CTBUILD is a wrapper around the gmake-based DØ build system. Its has less functionality than gmake or cmt, but it is more than adequate for 90% of the DØ packages. It degrades gracefully: if you need the raw power of gmake you can have it. But if a developer decides to do that, they must also commit to maintaining the make code (the build managers find it necessary to alter the underlying build system about once a year). Almost everyone uses the CTBUILD interface – experts and beginners alike.

This simplified approach could be applied to all aspects of particle physics: build, analysis, debugging, etc.

Conclusions

These modifications have been used in active DØ analysis (primarily the search for single top production). My contributions to that analysis would not have been as large if I did not have these tools available to me. Even debugging ROOT I/O code is easier with this system! The only difficulty with the tools was the integration with MSVC 2003, mostly caused by a very old extension model in MSVC. The most recently released version of MSVC contains a complete build engine for the first time, and Microsoft spent significant time improving the extension model. There is a possibility this code will be migrated to the new version of MSVC.