Part I
Build systems – what for?
Why?

- You write an application (source code) and need to:
  - Compile the source
  - Link to other libraries
  - Distribute your application as source and/or binary

- You would also love if you were able to:
  - Run tests on your software
  - Run test of the redistributable package
  - See the results of that

Compiling

- Manually?
  
gcc -DMYDEFINE -c myapp.o myapp.cpp

- Unfeasible when:
  - you have many files
  - some files should be compiled only in a particular platform, are you going to trust your brain?
  - different defines depending on debug/release, platform, compiler, etc

- You really want to automate this step
Linking

- Manually?
  
  `ld -o myapp file1.o file2.o file3.o -lc -lmylib`

- Again, unfeasiable if you have many files, dependence on platforms, etc

- You also want to automate this step

Distribute your software

- Traditional way of doing things:
  - Developers develop code
  - Once the software is finished, other people package it
  - There are many packaging formats depending on operating system version, platform, Linux distribution, etc: .deb, .rpm, .msi, .dmg, .src.tar.gz, .tar.gz, InstallShield, etc

- You'd like to automate this but, is it possible to bring packagers into the development process?
Testing

- You all use unit tests when you develop software, don't you? You should!
- When and how to run unit tests? Usually a three-step process:
  - You manually invoke the build process (e.g. make)
  - When it's finished, you manually run a test suite
  - When it's finished, you look at the results and search for errors and/or warnings
  - Can you test the packaging? Do you need to invoke the individual tests or the unit test manually?

Testing and gathering results

- Someone needs to do testing for each platform, then merge the results
- Is it possible to automate this? “make test”? what about gathering the results?
Your core business is software development, not software building

What are you selling?

- A flight simulator? or,
- A “flight simulator built with an awesome in-house developed built system”?

The client does not care about how you built your software, they are only interested in having the best software application possible

So should you: modern build systems should be able to build the software, package it, test it and tell

Part II
Build systems tour
Autotools

- It's been in use for many years and it's still widely used

- Autohell?
  - You need to write scripts in Bourne shell ('sh'), m4 (you all develop software in m4, don't you?),
  - Only Unix platform => Visual Studio, Borland, etc in Win32 are unsupported (Cygwin/MinGW supported)
  - Dependency discovery is mostly manual (no bundled “finders” grouping several steps)
  - Usually long, difficult to understand scripts
  - Autotools create a Makefile for 'make'

Jam

- The original implementation (Perforce Jam) is quite buggy
- There are many slightly different implementations
- Cross platform
- Dependency discovery is mostly manual (no bundled “finders” grouping several steps)
- Compiles and links by itself
- Users cannot use the tools they are used to
- What if Jam is not available for that platform?
- Allows parallel linking
SCons

Python DSL
- The interpreter is not always available
- You need to learn almost a programming language

Cross-platform
- You are actually writing a software app which happens to build another software app

Does not scale well
- Dependency discovery is mostly manual (no bundled “finders” grouping several steps)

Compiles and links by itself

Waf
- Second generation of bksys, tries to fix Scons
- No installation: it's a 100KB script you redistribute with your source
- It's a security issue: if a bug is found, every app needs to redistribute a new waf version
- Not cross-platform, won't ever be
- Recently added Win32
- Dependency discovery is mostly manual (you can write “finders” but you cannot reuse them)

Compiles and links by itself
CMake

- Cross-platform
- Very simple script language
- Dependency discovery is awesome:
  `FIND_PACKAGE`
- Scales very well: KDE4 is using it (4+ MLOC)
- Creates a project files for Makefile, Visual Studio, Kdevelop, Eclipse, etc
- Users can use the tools they are used to
  Cannot overcome the limitations those IDEs/'make' have

Part III
Meeting CMake
The Kitware build and test chain

- Cmake
- CPack
- CTest + BullsEye/gcov
- CDash

What is CMake

- Think of it as a meta-Make
- CMake is used to control the software compilation process using simple platform and compiler independent configuration files
- CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice: Visual C++, Kdevelop3, Eclipse, XCode, makefiles (Unix, NMake, Borland, Watcom, MinGW, MSYS, Cygwin), Code::Blocks etc
- Projects are described in CMakeLists.txt files (usually one per subdir)
Build flow

CMakeLists.txt

.cmake / CMakeSetup / CMakeGui

.vcproj / Makefile / etc

Native building tools (Visual Studio, Eclipse, KDevelop, etc)

.obj / .o

Native linking tools (lib.exe, link.exe, ld, etc)

.exe / .dll / .lib / .a / .so / .dylib

Tools the developer is already familiar with

In-source vs out-of-source

- Where to place object files, executables and libraries?

- In-source:
  - helloapp/hello.cpp
  - helloapp/hello.exe

- Out-of-source:
  - helloapp/hello.cpp
  - helloapp-build/hello.exe

- CMake prefers out-of-source builds
The CMake workflow

- Have this tree:
  - myapp
    - build
    - trunk
- cd myapp/build
- cmake ..../trunk
- make (Unix) or open project (VC++)
- On Windows, you can also use CMakeSetup (GUI). Cmake 2.6 includes a multiplatform Qt4-based GUI.

If Eclipse:
- myapp/trunk
- myapp-build
- Eclipse has problems if the build dir is a subdir of the source dir

Very simple executable

- PROJECT( helloworld )
- SET( hello_SRCS hello.cpp )
- ADD_EXECUTABLE( hello ${hello_SRCS} )

- PROJECT is not mandatory but you should use it
- ADD_EXECUTABLE creates an executable from the listed sources
- Tip: add sources to a list (hello_SRCS), do not list them in ADD_EXECUTABLE
Showing verbose info

- To see the command line CMake produces:
  - `SET(CMAKE_VERBOSE_MAKEFILE on)`
  - Or:
    - `$ make VERBOSE=1`
  - Or:
    - `$ export VERBOSE=1`
    - `$ make`
- Tip: only use it if your build is failing and you need to find out why

Very simple library

- `PROJECT(mylibrary)`
- `SET(mylib_SRCS library.cpp)`
- `ADD_LIBRARY(my SHARED ${mylib_SRCS})`

- `ADD_LIBRARY` creates an static library from the listed sources
- Add `SHARED` to generate shared libraries (Unix) or dynamic libraries (Windows)
Shared vs static libs

- Static libraries: on linking, add the *used* code to your executable
- Shared/Dynamic libraries: on linking, tell the executable where to find some code it needs
- If you build shared libs in C++, you should also use soversioning to state binary compatibility (too long to be discussed here)

The CMake cache

- Cmake is very fast on Unix but noticeably slow on Windows with Microsoft Visual C++ due to VC++ slowness to check types
- The CMake cache stores values which are not usually changed
- Edit the cache using ccmake (Unix) or CMakeSetup (Windows)
Variables & cache (I)

- Unconditional set
- \texttt{SET( var1 13 )}
  - "var1" is set 13
  - If “var1” already existed in the cache, it is shadowed by this value
  - This call does not overwrite “var1” value in the cache, if it existed

Variables & cache (II)

- Reuse the cache
- \texttt{SET( var2 17 ... CACHE ... )}
  - “var2” already in cache => keep cache value
  - “var2” not yet in cache (usually during first cmake run) => var2 is set to 17 and this goes into the cache
  - The value in the cache can be changed by editing \texttt{CMakeCache.txt}, or "make edit_cache", or running ccmake or running cmake-gui.
Variables & cache (III)

- Unconditional set & overwrite cache
- SET(var3 23 ... CACHE FORCE)
  - “var3” always takes this value, whether it was already in the cache or not
  - Cached value will always be overwritten => this makes editing the cache manually impossible

Regular expressions

- Worst side of Cmake: they are non-PCRE
- Use STRING( REGEX MATCH ... ), STRING (REGEX MATCHALL ... ), STRING(REGEX REPLACE ... )
- You will need to try once and again until you find the right regex
- I'm implementing STRING( PCRE_REGEX MATCH ...), etc based on PCRE. Not sure if it will be on time for Cmake 2.6.0 – It won't be
Back/Forward compatibility

- Since Cmake 2.0, ask for at least a certain version with CMAKE_MINIMUM_REQUIRED
- Since Cmake 2.6, tell Cmake to behave like a certain version ( > 2.4) with CMAKE_POLICY(VERSION major.minor[.patch] )

Part IV
Real world CMake:
dependencies between targets
Adding other sources

- clockapp
- build
- trunk
- doc
- img
- libwakeup
  - wakeup.cpp
  - wakeup.h
- clock
  - clock.cpp
  - clock.h

```
PROJECT(clockapp)
ADD_SUBDIRECTORY(libwakeup)
ADD_SUBDIRECTORY(clock)

SET(wakeup_SRCS
  wakeup.cpp)
ADD_LIBRARY(wakeup SHARED
  ${wakeup_SRCS})

SET(clock_SRCS clock.cpp)
ADD_EXECUTABLE(clock $ {clock_SRCS})
```

Variables

- No need to declare them
- Usually, no need to specify type
- SET creates and modifies variables
- SET can do everything but LIST makes some operations easier
- Use SEPARATE_ARGUMENTS to split space-separated arguments (i.e. a string) into a list (semicolon-separated)
- In Cmake 2.4: global (name clashing problems)
- In Cmake 2.6: scoped
Changing build parameters

- Cmake uses common, sensible defaults for the preprocessor, compiler and linker
- Modify preprocessor settings with 
  ADD_DEFINITIONS and REMOVE_DEFINITIONS
- Compiler settings: CMAKE_C_FLAGS and CMAKE_CXX_FLAGS variables
- Tip: some internal variables (CMAKE_) are read-only and must be changed executing a command

Flow control

- IF(expression) ...
  ELSE(expression) ...
 ENDIF(expression)
- Process a list:
  FOREACH(loop_var) ...
  ENDFOREACH(loop_var)
- WHILE(condition) ...
  ENDWHILE(condition)

Always repeat the expression/condition
It's possible to avoid that but I won't tell you how
To show .h files in Visual Studio, add them to the list of sources in `ADD_EXECUTABLE` / `ADD_LIBRARY`

```cmake
SET(wakeup_SRCS wakeu.cpp)
IF(WIN32)
  SET(wakeup_SRCS ${wakeup_SRCS}
wakeup.h)
ENDIF(WIN32)
ADD_LIBRARY(wakeup SHARED ${wakeup_SRCS})
```

Use `SOURCE_GROUP` if all your sources are in the same directory

Managing debug and release builds

```cmake
SET(CMAKE_BUILD_TYPE Debug)
```

As any other variable, it can be set from the command line:

```
cmake -DCMAKE_BUILD_TYPE=Release ../trunk
```

Specify debug and release targets and 3rdparty libs:

```cmake
TARGET_LINK_LIBRARIES(wakeup RELEASE ${wakeup_SRCS})
TARGET_LINK_LIBRARIES(wakeupd DEBUG ${wakeup_SRCS})
```
Standard directories... not!

- Libraries built in your project (even if in a different CmakeLists.txt) is automatic (in rare occasions: ADD_DEPENDENCIES)
- If the 3rd party library or .h is in a “standard” directory (PATH and/or LD_LIBRARY_PATH) is automatic
- If in a non-standard dir:
  - Headers: use INCLUDE_DIRECTORIES
  - Libraries: use FIND_LIBRARY and link with the result of it (try to avoid LINK_DIRECTORIES)

make install

- INSTALL(TARGETS clock wakeup RUNTIME DESTINATION bin LIBRARY DESTINATION lib)
- Would install in /usr/local/bin and /usr/local/lib (Unix) or %PROGRAMFILES%\projectname (Windows)
Part V
Platform checks and external dependencies

Finding installed software

- `FIND_PACKAGE(Qt4 REQUIRED)`
- Cmake includes finders (FindXXXX.cmake) for ~130 software packages, many more available in Internet
- If using a non-CMake FindXXXX.cmake, tell Cmake where to find it by setting the CMAKE_MODULE_PATH variable
- Think of FIND_PACKAGE as an #include
Qt with CMake

PROJECT( pfrac )
FIND_PACKAGE( Qt4 REQUIRED )
INCLUDE( ${QT_USE_FILE} )

SET( pfrac_SRCS main.cpp client.h client.cpp )
SET( pfrac_MOC_HEADERS client.h )

QT4_ADD_RESOURCES( pfrac_SRCS
  ${PROJECT_SOURCE_DIR}/pfrac.qrc )
QT4_WRAP_CPP( pfrac_MOC_SRCS
  ${pfrac_MOC_HEADERS} )

ADD_EXECUTABLE( pfrac ${pfrac_SRCS} $
{pfrac_MOC_SRCS}
TARGET_LINK_LIBRARIES( pfrac ${QT_LIBRARIES} )

Platform includes

- CONFIGURE_FILE(InputFile OutputFile
  [COPYONLY] [ESCAPE_QUOTES] [@ONLY])

  - Your source may need to set some options depending on the platform, build type, etc

  - Create a wakeup.h.cmake and:
    - #cmakedefine VAR will be replaced with #define VAR if VAR is true, else with /* #undef VAR */
    - @VAR@ will be replaced with the value of VAR

  - Also useful for .conf files
Platform includes (II)

- CHECK_TYPE_SIZE (needs INCLUDE(CheckTypeSize))
- TEST_BIG_ENDIAN (needs INCLUDE(CheckBigEndian))
- CHECK_INCLUDE_FILES (needs INCLUDE(CheckIncludeFiles))

Platform Includes (III)

CMakeLists.txt

```cpp
#include "wakeup.h"
#include "wakeup2.h"
#ifdef HAVE_MALLOC_H
#include <malloc.h>
#else
#include <stdlib.h>
#endif
dvoid do_something()
```
Part VI
Macros and functions

Macros

- MACRO( <name> [arg1 [arg2 [arg3 ...]]] )
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
  ENDMACRO( <name> )

- They perform text substitution, just like \texttt{#define} does in C

- Danger! Variable-name clashing is possible if using too generic names. Hint: prefix your varnames with the macro name: \texttt{MACRO\_VARNAME} instead of \texttt{VARNAME}
Functions

- New in Cmake 2.6
- Real functions (like C), not just text-replace (a-la C preprocessor)
- Advantages: avoid variable-scope trouble (hopefully)

New targets

- Targets defined with ADD_CUSTOM_TARGET are always considered outdated (i.e. rebuilt)
- Two signatures for ADD_CUSTOM_COMMAND:
  - Same as ADD_CUSTOM_TARGET but do not rebuild if not needed
  - Execute a target before build, after build or before link
- For example, you can create GENERATE_DOCUMENTATION
MACRO(GENERATE_DOCUMENTATION DOXYGEN_CONFIG_FILE)
FIND_PACKAGE(Doxygen)
SET(DOXYFILE_FOUND false)
IF(EXISTS \$PROJECT_SOURCE_DIR\\$/DOXYGEN_CONFIG_FILE)
SET(DOXYFILE_FOUND true)
ENDIF(EXISTS \$PROJECT_SOURCE_DIR\\$/DOXYGEN_CONFIG_FILE)

IF( DOXYGEN_FOUND )
    IF( DOXYFILE_FOUND )
        # Add target
        ADD_CUSTOM_TARGET( doc ALL \$DOXYGEN_EXECUTABLE \$PROJECT_SOURCE_DIR\\$/DOXYGEN_CONFIG_FILE"
        # Add .tag file and generated documentation to the list of files we must erase when distcleaning
        # Read doxygen configuration file
        FILE( READ \$PROJECT_SOURCE_DIR\\$/DOXYGEN_CONFIG_FILE
              DOXYFILE_CONTENTS )
        STRING( REGEX REPLACE \"\n \" DOXYFILE_LINES
              $DOXYFILE_CONTENTS )
...

# Parse .tag filename and add to list of files to delete if it exists
FOREACH( DOXYLINE \$DOXYFILE_CONTENTS )
    STRING( REGEX REPLACE \".*GENERATE_TAGFILE *= *([\^\n]+).\" DOXYGEN_TAG_FILE \$DOXYLINE )
ENDFOREACH( DOXYLINE )
ADD_TO_DISTCLEAN( \$PROJECT_BINARY_DIR\\$/DOXYGEN_TAG_FILE )

# Parse doxygen output doc dir and add to list of files to delete if it exists
FOREACH( DOXYLINE \$DOXYFILE_CONTENTS )
    STRING( REGEX REPLACE \".*OUTPUT_DIRECTORY *= *([\^\n]+).\" DOXYGEN_DOC_DIR \$DOXYLINE )
ENDFOREACH( DOXYLINE )
ADD_TO_DISTCLEAN( \$PROJECT_BINARY_DIR\\$/DOXYGEN_DOC_DIR )
ADD_TO_DISTCLEAN( \$PROJECT_BINARY_DIR\\$/DOXYGEN_DOC_DIR.dir )
...

# Parse doxygen output doc dir and add to list of files to delete if it exists
FOREACH( DOXYLINE \$DOXYFILE_CONTENTS )
    STRING( REGEX REPLACE \".*OUTPUT_DIRECTORY *= *([\^\n]+).\" DOXYGEN_DOC_DIR \$DOXYLINE )
ENDFOREACH( DOXYLINE )
ADD_TO_DISTCLEAN( \$PROJECT_BINARY_DIR\\$/DOXYGEN_DOC_DIR )
ADD_TO_DISTCLEAN( \$PROJECT_BINARY_DIR\\$/DOXYGEN_DOC_DIR.dir )
...
ELSE( DOXYFILE_FOUND )
MESSAGE( STATUS "Doxygen configuration file not found - Documentation will not be generated" )
ENDIF( DOXYFILE_FOUND )
ELSE(DOXYGEN_FOUND)
MESSAGE(STATUS "Doxygen not found - Documentation will not be generated")
ENDIF(DOXYGEN_FOUND)
ENDMACRO(GENERATE_DOCUMENTATION)

Calling the outside world

- EXECUTE_PROCESS
- Execute and get output from a command, copy files, remove files, etc
- Cross-platform: avoid calling /bin/sh or cmd.exe if EXECUTE_PROCESS suffices
Part VII
Creating your own finders

What is a finder

- When compiling a piece of software which links to third-party libraries, we need to know:
  - Where to find the .h files (\texttt{-I} in gcc)
  - Where to find the libraries (.so/.dll/.lib/.dylib/...) (\texttt{-L} in gcc)
  - The filenames of the libraries we want to link to (\texttt{-l} in gcc)
- That's the basic information a finder needs to return
MESSAGE

- Show status information, warnings or errors
  
  \[
  \text{MESSAGE( } [\text{SEND\_ERROR} \mid \text{STATUS} \mid \text{FATAL\_ERROR}] \\
  \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{"message to display" } \ldots \text{ )}
  \]

STRING

- Manipulate strings or regular expressions
- Many signatures
Files and Windows registry

GET_FILENAME_COMPONENT interacts with the outside world

- Sets a Cmake variable to the value of an environment variable
- Gets a value from a Windows registry key
- Gets basename, extension, absolute path for a filename

FILE

Read from / write to files
Remove files and directories
Translate paths between native and Cmake: \\ ↔ /
Find libraries

- `FIND_LIBRARY` and the `CMAKE_LIBRARY_PATH` variable

- (this slide is only a stub)

Find header files

- `FIND_FILE`

- (this slide is only a stub)
Find generic files

- `FIND_PATH` and the `CMAKE_INCLUDE_PATH` variable
- (this slide is only a stub)

PkgConfig support

- PkgConfig is a helper tool used when compiling applications and libraries
- PkgConfig provides the `-L`, `-l` and `-I` parameters
- Try to avoid it, as it's not always installed
- Mostly Unix, available for Win32 but seldomly used
- Cmake provides two paths to use PkgConfig: `UsePkgConfig.cmake` and `FindPkgConfig.cmake`
- FIND_PROGRAM
  - (this slide is only a stub)

- TRY_COMPILE
  - (this slide is only a stub)
- TRY_RUN
- (this slide is only a stub)

Part VIII
Properties
- CMAKE_MINIMUM_REQUIRED
  (this slide is only a stub)

- OPTION
  (this slide is only a stub)
- GET_CMAKEPROPERTY
  (this slide is only a stub)

- GET_TARGETPROPERTY
  (this slide is only a stub)
- SET_TARGET_PROPERTIES
- (this slide is only a stub)

- SET_SOURCE_FILES_PROPERTIES
- (this slide is only a stub)
Part IX
Useful variables

- \texttt{CMAKE_BINARY_DIR/CMAKE_SOURCE_DIR}
- (this slide is only a stub)
- CMAKE_CURRENT_BINARY_DIR
  /
CMAKE_CURRENT_SOURCE_DIR
- (this slide is only a stub)

- PROJECT_BINARY_DIR/PROJECT_SOURCE_DIR
- (this slide is only a stub)
- EXECUTABLE_OUTPUT_PATH/LIBRARY_OUTPUT_PATH
- (this slide is only a stub)

- ENV($ENV{name})
- (this slide is only a stub)
- **CMAKE_SKIP_RPATH** (important in Debian and Debian-derivatives) (follow 
  [http://www.cmake.org/Wiki/CMake_RPATH_handling](http://www.cmake.org/Wiki/CMake_RPATH_handling))

- (this slide is only a stub)

---

**More variables**

- Use this snippet to list all variables and their values:

```cpp
get_cmake_property( P VARIABLES )
foreach( VAR in ${P} )
  message( STATUS
           " ${VAR}=${${VAR}}" )
endforeach()
```
Part X
CPack

Features

- CPack generates installing packages:
  - RPM, DEB, GZip and Bzip2 distributions of both binaries and source code
  - NSIS installers (for Microsoft Windows)
  - Mac OS X packages (.dmg)
  - In Cmake 2.4, .rpm and .deb support works but is not good
  - It can be used without Cmake
  - If used with Cmake, takes advantage of the INSTALL declarations
Variables in CPack

- There are bundle-specific variables: NSIS needs some vars a ZIP does not need
- Important: set variable values BEFORE you INCLUDE( CPack )

Example

INCLUDE(InstallRequiredSystemLibraries)

SET(CPACK_PACKAGE_DESCRIPTION_SUMMARY "Alarm clock")
SET(CPACK_PACKAGE_VENDOR "Pau Garcia i Quiles")
SET(CPACK_PACKAGE_DESCRIPTION_FILE
"$CMAKE_CURRENT_SOURCE_DIR}/ReadMe.txt")
SET(CPACK_RESOURCE_FILE_LICENSE
"$CMAKE_CURRENT_SOURCE_DIR}/Copyright.txt")
SET(CPACK_PACKAGE_VERSION_MAJOR "0")
SET(CPACK_PACKAGE_VERSION_MINOR "0")
SET(CPACK_PACKAGE_VERSION_PATCH "1")
SET(CPACK_PACKAGE_INSTALL_DIRECTORY "CMake ${Cmake_VERSION_MAJOR}.${Cmake_VERSION_MINOR}")

...
IF(WIN32 AND NOT UNIX)
SET(CPACK_PACKAGE_ICON "$
{Cmake_SOURCE_DIR}/Utilities/Release\\\InstallIcon.bmp")
SET(CPACK_NSIS_INSTALLED_ICON_NAME
"bin\\\\MyExecutable.exe")
SET(CPACK_NSIS_DISPLAY_NAME "$
{CPACK_PACKAGE_INSTALL_DIRECTORY} My Famous Project")
SET(CPACK_NSIS_HELP_LINK "http:\\\\\\elpauer.org")
SET(CPACK_NSIS_URL_INFO_ABOUT "http:\\\\\\elpauer.org")
SET(CPACK_NSIS_CONTACT "pgquiles@elpauer.org")
...

INCLUDE(CPack)
Features

- Cross-platform testing system which:
  - Retrieves source from CVS, Subversion or Perforce (git support currently being worked on)
  - Configures and build the project
  - Configures, build and runs a set of predefined runtime tests
  - Sends the results to a Dart/CDash dashboard

- Other tests:
  - code coverage: using BullsEye ($$$) or gcov (free)
    (note to self: show rbxspf code coverage)
  - memory checking

Example

- Very easy!
  - ENABLE_TESTING()
  - ADD_TEST( testname testexecutable args )

- Some scripting needed to:
  - Download sources from a VC system (CVS, SVN and Perforce templates available, git in progress)
  - Upload to Dart/CDash dashboard (templates available for HTTP, FTP, SCP and XML-RPC)

- It can be used with non-CMake projects
Part XII
CDash

Features

- CDash aggregates, analyzes and displays the results of software testing processes submitted from clients.
- Replaces Dart
- For example, build a piece of software on Linux, Windows, Mac OS X, Solaris and AIX
- Usually, you want two kinds of information:
  - Build results on all platforms
  - Test (Ctest) results on all platforms
- Customizable using XSL
### Nightly Changes as of 2008-03-28 01:00:00 EDT

<table>
<thead>
<tr>
<th>Site</th>
<th>Build Name</th>
<th>Update</th>
<th>Cfg</th>
<th>Build</th>
<th>Test</th>
<th>Build Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>purple kiddare</td>
<td>darwin-2014 0.1</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>2008-03-28 02:22:00 EDT</td>
</tr>
<tr>
<td>lw furry</td>
<td>Linux-x86 1-rel-static</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2008-03-28 07:22:00 EDT</td>
</tr>
<tr>
<td>lw panther</td>
<td>macOSX-gcc4 2-rel-static</td>
<td>0</td>
<td>0</td>
<td>35</td>
<td>0</td>
<td>2008-03-28 03:36:00 EDT</td>
</tr>
</tbody>
</table>

#### No Continuous Builds

#### No Experimental Builds

#### No Coverage

#### No Dynamic Analysis