FreeRDP Developer Manual

Marc-André Moreau

February 17, 2013

Contents

Chapter 1

Introduction

1.1 Glossary

To be expanded.

1.2 References

To be expanded.

Chapter 2

Build Environment

2.1 CMake Build System

FreeRDP uses CMake as its build system. CMake is not an ordinary build system: it generates project files for other build systems to use. This means that we have only one set of scripts to maintain while having the flexibility of using any of the build systems supported by CMake. For instance, FreeRDP developers can use Visual Studio, Xcode, Eclipse, or just plain regular makefiles without having to separately maintain project files for the development tools they are using.

2.1.1 Installation

FreeRDP requires CMake 2.8, but it is highly recommended to use the very latest builds available from the CMake website (www.cmake.org). This is because certain useful features are available only in certain versions that we cannot require since they are too recent. Using an older version of CMake 2.8 will work, but some features taking advantage of functionality present in more recent CMake releases may be disabled. For instance, FreeRDP offers a monolithic build option that combines all libraries into a single one using functionality found in CMake 2.8.9. If you are using CMake 2.8.8, this option will be disabled.

Recent CMake builds are available from the CMake download page:

http://www.cmake.org/cmake/resources/software.html

Linux

CMake is packaged by most Linux distributions and should be easy to install. However, since FreeRDP leverages some of the newest CMake features, it is very

likely that the version packaged for your system is not the latest one. In this case, simply download the CMake Linux build from the CMake website and install it manually to the appropriate location.

Mac OS X

The easiest way to install CMake on Mac OS X is by using the latest build available on the CMake website. Alternatively, one can use common Mac OS X package manager such as homebrew to install CMake.

Windows

Installing CMake on Windows is fairly straightforward. Download either the installer of the zip version of the Windows build from the CMake website. Using the installer is simpler, but you might want to consider choosing the zip archive if you need to work from a machine where you do not have sufficient rights to run an installer.

2.2 Git Version Control

FreeRDP is hosted on GitHub, and uses git for version control. Git is one of those tools that is extremely powerful but a bit hard to grasp at first. Luckily, a lot of the complexity of git is made much simpler through the GitHub collaborative coding interface, and provides us with a functional workflow for accepting contributions with minimal overhead.

If you are new to FreeRDP and don't want to bother learning the git basics for now, feel free to simply download a source tarball from GitHub and move on to the next step. However, you will eventually have to learn it, especially if you want to synchronize your work with the latest sources, or contribute your changes back to the community. If you are not so sure about how to contribute changes which have not been done cleanly with git, ping one of the developers in the IRC channel. It's never fun to get contributions in the form of tarball in an email, but we'll take it anyway. Please understand, however, that since this require much more effort on our side, clean integration of the changes will only happen when time allows for it.

2.2.1 Installation

Git is packaged for most Linux distributions and should be easily obtained. Otherwise, and for all other platforms, pre-built packages are available from the Git download page (http://git-scm.com/downloads)

Many other third-party Git packages exist for various operating systems. On Windows, git can be installed in Cygwin http://www.cygwin.com/, or through msysgit http://msysgit.github.com/

2.2.2 Configuration

Configuring Git can be tricky, especially for the part where one has to generate SSH keys to be used for authentication with GitHub instead of passwords.

One does not necessarily need to use GitHub, but unless you know what you are doing you should simply create a GitHub account and follow the regular GitHub workflow.

GitHub offers extensive help which everyone should refer to: https://help.github.com/

Alternatively, the Pro Git book is available for free online: http://git-scm.com/book

The Git Reference is concise, small and to the point: http://gitref.org/

Along with the GitHub help, there is an excellent step-by-step introduction to GitHub on the GitHub Learn website: http://learn.github.com/p/setup.html

If you need more assistance getting started, please come on IRC and we'll help you.

2.3 Native Compilation

2.3.1 Linux

In debian-based distributions, the build-essential package installs GCC and common development tools necessary for C/C++ development.

2.3.2 Mac OS X

Xcode is the easiest way develop for Mac OS X, and is distributed for free on the Mac App Store.

The Xcode command line tools are not installed by default with Xcode and are necessary:

- On the Xcode menu, click Xcode and then Preferences
- Select the Downloads tab from the Preferences panel
- Select the Components sub-tab from the Downloads tab
- Select Command line tools and click install on the right

2.3.3 Windows

FreeRDP supports all versions of Windows starting from Windows XP all the way up to Windows 8. Windows development is normally done using Visual Studio 2010 or Visual Studio 2012. If you do not own a license of Visual Studio, the Express edition is available for free and should work fine.

To build FreeRDP on Windows, you will first need to install OpenSSL (see the "Prerequisites" section), along with the rest of the Windows build environment, including CMake. When installing CMake, it is recommended to select the option that adds it to the system path.

Refer to the "Build Configuration" section to learn the basic usage of CMake. For Windows, common CMake generators are "Visual Studio 2010" and "Visual Studio 2012". There is also the "NMake Makefiles" generator which may be suitable for more advanced Windows developers who would prefer a build system that does not require the full Visual Studio IDE.

2.4 Cross-Compilation

2.4.1 Android

Android development can be done from Linux, Mac OS X and Windows. There are two development kits for Android: the Software Development Kit (SDK) and the Native Development Kit (NDK). The SDK is for Java, while the NDK is for C/C++. Since it is not possible to write completely native Android applications (at least not under normal circumstances), we will need both the Android SDK and NDK for FreeRDP development. The NDK alone is enough to build the FreeRDP libraries, but not enough to build a complete application.

Java Development Kit

Before installing the SDK, you will need to install the Java Development Kit. On Windows, the Sun/Oracle JDK is easy to download and install. On Linux, OpenJDK is easier to install from your distribution's package manager.

Android SDK and NDK

Download the Android SDK and Android NDK from the Android developer website:

http://developer.android.com/tools/

Extract or install the SDK and NDK to a path which is easy to locate. The Windows installer installs the SDK into "%PROGRAMFILES(x86)%/Android" by

default. On Linux, using /opt/android-sdk and /opt/android-ndk is common practice. Regardless of where you choose to install the SDK and NDK, you will need to configure the ANDROID_SDK and ANDROID_NDK environment variables to point to them. It is also very convenient to add $ANDROID_SDK/tools and $ANDROID_SDK/platform-tools to your system path in order to have the Android utilities available without specifying the full path.

Once both the SDK and NDK are installed, the first thing to do (it takes some time) is to launch the "android" utility found in $ANDROID_SDK/tools. On Windows, the same utility can be launched with the "SDK Manager" found in %ANDROID_SDK%. In the SDK manager, select target Android platforms of interest, such as Android 2.3.3 (API 10) or Android 4.0 (API 14), click install, accept the licenses, and be patient because it takes time to download and install everything. The good news is that you can leave the SDK manager open while moving on to setting up the NDK.

Environment Variables

You will need to set a certain number of environment variables to ease Android development. On Linux, this can be done by editing the .bashrc file, and on Mac OS X, this can be done by editing the .profile file.

export NDK=/opt/android-ndk

export ANDROID_NDK=$NDK

export ANDROID_SDK=/opt/android-sdk

export PATH=$ANDROID_SDK/tools$:ANDROID_SDK/platform-tools:$PATH

Preparing Toolchain

The prebuilt toolchains that come with the Android NDK should suffice, but if you need to build your own, invoke the make-standalone-toolchain.sh script:

$NDK/build/tools/make-standalone-toolchain.sh --platform=android-8 --install-dir=$ANDROID_STANDALONE_TOOLCHAIN

Where $ANDROID_STANDALONE_TOOLCHAIN is the location where you want to install your standalone toolchain.

Invoking CMake

When invoking CMake, the path to the Android CMake toolchain needs to be passed:

cmake -DCMAKE_TOOLCHAIN_FILE=cmake/AndroidToolchain.cmake .

The Android CMake toolchain will make use of the environment variables defined in the previous section, so make sure they are correct before invoking CMake.

The Android CMake toolchain included with FreeRDP is based on the android-cmake project:

https://code.google.com/p/android-cmake/

Android Project

While the native portion of the FreeRDP Android port is built using CMake, the Android Java code is built using ant or Eclipse with the ADT plugin. The Android project structure is explained here:

http://developer.android.com/tools/projects/

This project is generated and maintained using the android command-line tool:

http://developer.android.com/tools/projects/projects-cmdline.html

The root of the Android project tree is located in client/Android:

src: android Java code (built with ant or Eclipse)

bin: build output directory (.apk files)

jni: android native code (built with cmake)

gen: android generated code such as R.java

assets: project assets, added as-is to the .apk

res: contains XML configuration files for menus, layouts, strings, etc

libs: contains third-party libraries for both Java and native code

Since the Android native code is built with cmake from the top-level directory, there is a small issue where the library output path does not match the one from the Android project. CMake outputs libfreerdp-android.so to libs/armeabi-v7a, but the Android project expects them in client/Android/libs/armeabi-v7a. You can either copy libfreerdp-android.so manually every time, or create a symlink to work around the issue:

cd client/Android/libs

ln -s ../../../libs/armeabi-v7a/ armeabi-v7a

In order to verify that libfreerdp-android.so gets properly packaged in the .apk, simply unzip the resulting .apk files to see its contents (an .apk file is a regular zip file with a different extension).

Ant Building

Inside client/Android, invoke ant help to list targets:

ant help

Targets of interest are debug and release, which generate a debug or release .apk file in the bin directory:

ant debug

ant release

The resulting .apk files can be deployed to actual devices or the Android emulator.

Eclipse Building

There is an Eclipse project located in client/Android, which is different from any potential Eclipse project generated by CMake in the root of the FreeRDP source tree. It can be imported in Eclipse like any other regular Eclipse project. Make sure to configure the Eclipse ADT plugin properly beforehand.

2.4.2 iOS

iOS development requires Xcode and the same development environment as for Mac OS X with the addition of the iOS SDK. Certain features like deploying to real devices are limited to Apple developers only.

Invoking CMake

FreeRDP includes a modified version of the ios-cmake toolchain in the cmake directory. The original toolchain can be found here:

http://code.google.com/p/ios-cmake/

When invoking CMake for iOS development, the iOS toolchain needs to be passed like this:

cmake -DCMAKE_TOOLCHAIN_FILE=cmake/iOSToolchain.cmake .

2.5 Prerequisites

The general approach towards prerequisites is simple: avoid hard dependencies as much as possible, make soft dependencies modular and optional. This may appear a bit extreme, but it is not when FreeRDP needs to run on a very large

variety of platforms with a small footprint. Convenience always comes at a cost, and we have learned from out past mistakes that in our case the extra effort is worth it.

2.5.1 OpenSSL

FreeRDP's only strong dependency is OpenSSL. All other dependencies can be made optional with the side effect of disabling the modules that depend on it. It is within our plans to make FreeRDP less dependent on OpenSSL in the future through the WinPR project: instead of using OpenSSL directly, we will implement the equivalent portions of what we need from the Windows API. The WinPR implementation of the Windows API will make use of OpenSSL and eventually add support for other cryptographic libraries. Not only will this enable OpenSSL-free builds of FreeRDP on Windows, but it will also give us an abstraction layer over OpenSSL, allowing us to add support for alternative libraries such as NSS, PolarSSL and CyaSSL.

OpenSSL is used for TLS, MD4, MD5, HMAC-MD5, SHA1, HMAC-SHA1, RSA, DES3, Base64, BigNum arithmetic and certificate validation. RDP8 support will require DTLS which is also provided by OpenSSL.

Linux

OpenSSL is packaged by most Linux distributions, and is easy to compile from source.

Mac OS X

OpenSSL used to ship with Mac OS X until Mac OS X 10.8 (Mountain Lion). Luckily, OpenSSL ships with Xcode. Otherwise, OpenSSL can be obtained from common Mac OS X package managers such as macports.

Command-line tools are not installed by default with Xcode and are necessary.

Windows

A binary distribution of OpenSSL is maintained by "The Shining Light Productions":

http://slproweb.com/products/Win32OpenSSL.html

In order to use it, download both the Win32 OpenSSL installer (the full one, not the light) and the Visual C++ 2008 Redistributables. Install the Visual C++ redistributables first, since OpenSSL requires it. Install OpenSSL in the default path suggested by the installer (C:\OpenSSL-Win32), otherwise may not find

it automatically. If you install OpenSSL in a path not found by CMake, you will need to use the OPENSSL_ROOT_DIR option to tell CMake where to find it.

Compilation The Visual C++ 2008 Redistributable requirement is a problem if you want to ship FreeRDP for Windows without an installer. Unfortunately, this requires building OpenSSL manually and modifying the default build scripts to link against the static Visual Studio runtime (/MT switch as opposed to the default /MD switch used by OpenSSL). There are many pitfalls along the way if you want to make a proper build of FreeRDP and OpenSSL using the static Visual Studio runtime.

A modified version of OpenSSL for Windows is available for the sake of convenience here:

https://github.com/awakecoding/openssl-win32

The sources were deliberately put on git to allow others to consult the history of modifications from the vanilla sources that allow for an easier Windows build.

In order to build OpenSSL, you will need to install the following prerequisites

ActivePerl (http://www.activestate.com/activeperl) NASM for Windows (http://www.nasm.us/) Microsoft Visual Studio (2010 or 2012) Raspberry Perl can be used instead of ActivePerl, and other versions of Visual Studio can be used, but the build scripts provided will need to be modified accordingly.

Once prerequisites are installed, download the modified OpenSSL sources from the openssl-win32 GitHub repository. In the root of the openssl-win32 directory, you should find two batch files: win32_env.bat and win32_build.bat. Edit both of them to ensure that the path variables they use are correct for your environment.

Once the batch files are properly edited, open a command prompt and move to the root of the openssl-win32 directory. Run win32_env.bat once in order to configure environment variables. Run win32_build.bat in order to build OpenSSL. When building is complete, you should find the resulting build in the "build" directory. This directory can be renamed to OpenSSL-Win32 and moved to C:\OpenSSL-Win32 to be automatically picked up by CMake. Beware, however, that this build has only been tested at this point with FreeRDP release builds configured with the MSVC_RUNTIME="static" option.

If your FreeRDP builds start crashing for no obvious reason when freeing memory, this is most likely the side effect of an MSVC runtime mismatch. Don't waste your time trying to find problems where they are not: even if you think you've compiled everything correctly chances are that you're fighting with the pickiness of OpenSSL builds on Windows.

Android

Even though OpenSSL is used in Android, the headers and libraries are not part the Android NDK. This means that in order to compile FreeRDP for Android, one must first compile OpenSSL manually.

Android uses its own build system which is not supported by OpenSSL out of the box. A modified version of OpenSSL with Android build scripts is available through the Android Open Source Project (AOSP). This is the library which is used for the Android operating system itself, but not made available to application developers through the NDK.

This may look like a good deal, but further modifications to default OpenSSL Android port are required before it can be built for usage with FreeRDP, such as adding an option for a static build.

At the time of writing this, a usable OpenSSL sources can be found from "The Guardian Project":

https://github.com/guardianproject/android-external-openssl-ndk-static

Just in case the original sources are taken down, here is a backup repository:

https://github.com/awakecoding/android-external-openssl-ndk-static

Download the modified OpenSSL sources, open a terminal, and move to the root of the source tree. Ensure you have the proper environment variables configured for Android development as covered earlier, and type the following command:

$NDK/ndk-build

Copy the OpenSSL headers and library files to your target android platform directory:

export ANDROID_PLATFORM=android-8

cp -R include/ $NDK/platforms/$ANDROID_PLATFORM/arch-arm/usr/include/

cp obj/local/armeabi/*.a $NDK/platforms/$ANDROID_PLATFORM/arch-arm/usr/lib/

iOS

There is no pre-built OpenSSL iOS package, so again we have to build our own. The "iOSPorts" collection of iOS ports simplifies the task of building OpenSSL for iOS.

Download the iOSPorts collection from: https://github.com/bindle/iOSPorts

Launch Terminal and move to the root of the iOSPorts source tree. From there, type the following commands:

cd ports/security/openssl/

make

cd openssl

../build-aux/configure-wrapper.sh

make install

This will download, patch, configure build and install OpenSSL for iOS. The default installation path is in /tmp/iOSPorts. The important files are the include and lib directories from the OpenSSL, which we will need to copy to a location inside the iOS SDK.

The system root for the iOS SDK can be found in the following path:

/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS6.0

Where "iPhoneOS6.0.sdk" is subject to change depending on the SDK version.

To ease installation, configure the following environment variables:

export IOS_SDK_ROOT=/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Deve

export OPENSSL_ROOT=/tmp/iOSPorts

Then enter the following commands to install OpenSSL in the iOS SDK:

cp -R "$OPENSSL_ROOT/include$""IOS_SDK_ROOT/usr"

cp "$OPENSSL_ROOT/lib/libssl.a$""IOS_SDK_ROOT/usr/lib/libssl.a"

cp "$OPENSSL_ROOT/lib/libcrypto.a$""IOS_SDK_ROOT/usr/lib/libcrypto.a"

This procedure only has to be executed once per iOS SDK. After this, OpenSSL should be picked up automatically by CMake.

2.5.2 X11

Linux

For the X11 client, you will need:

sudo apt-get install libx11-dev libxcursor-dev libxkbfile-dev libxinerama-dev

Additionally, for the X11 server, you will need:

sudo apt-get install libxext-dev libxtst-dev libxdamage-dev

Mac OS X

If the X11 SDK used to ship with XCode, it was removed from its newer versions. In order to get the X11 dependencies satisfied, one has to obtain it from the XQuartz open source project:

http://xquartz.macosforge.org/

Windows

Even if a native Windows port is available, it is possible to compile the X11 FreeRDP client on Windows using Cygwin. Please note, however, that this particular configuration is rarely tested.

Chapter 3

Build Configuration

Projects like FreeRDP need to conform to a large number of requirements at the same time while remaining highly flexible, portable, fast, small, modular, reusable, testable, configurable, and the list goes on. It may not be obvious, but one of the challenges of open source development is having to please everybody at the same time. Different vendors want FreeRDP built in certain ways, with nice extension points for themselves to easily maintain their extensions separately, with just about any option that fits their needs within the upstream sources. It turns out that providing everybody with what they want is possible with enough CMake scripting. Nevertheless, the task of offering build configuration for all these needs makes build configuration management quite complex.

3.1 CMake

Prior to generating build scripts for your environment, you should ensure that you have CMake installed and available within your system path. Installing CMake is covered in the previous section of this manual.

3.1.1 Introduction

To generate build scripts, open a terminal, move to the root of the source tree, and type:

cmake .

This will tell CMake to generate build scripts using the CMake scripts found in the local directory (“.”). If the command fails, you might need to invoke CMake with additional options which we will cover in the following sections. The path

to the directory where the root CMake script is contained is always the last argument of the cmake command.

CMake looks for files with the name CMakeLists.txt. These files are the equivalent of makefiles, written in the CMake scripting language. A root CMakeLists.txt script can include other such scripts from subdirectories.

3.1.2 Generators

CMake scripts are used to generate platform-specific build scripts with the help of CMake generator. When no generator is specified, CMake attempts finding a suitable default. If it can't find any, generation will fail. Most of the time, you will want to specify the generator of your choice using the –G "Generator Name" option. Here is a list of common generators:

- Xcode
- Unix Makefiles
- NMake Makefiles
- Visual Studio 11
- Visual Studio 10
- Visual Studio 9 2008
- Eclipse CDT4 – Unix Makefiles
- Eclipse CDT4 – NMake Makefiles

In order to generate an Eclipse project on Linux, one would type:

cmake –G "Eclipse CDT4 – Unix Makefiles" .

For simple makefiles without an Eclipse project, "Unix Makefiles" would do it.

On Windows, the generator would likely be "Visual Studio 10" (Visual Studio 2010)

On Mac OS X, the "Xcode" generator would be appropriate.

3.1.3 CMake Cache

When CMake executes, it generates CMakeCache.txt, or the CMake cache. The cache stores values from previous executions. Whenever major changes happen, such as installing new dependencies, modifying the CMake scripts, or when something does not work as expected it is advised to simply delete the CMake

cache and regenerate from scratch. Clearing the CMake cache is done by deleting the CMakeCache.txt file:

rm CMakeCache.txt

The CMake cache is a simple text file which looks like a simple configuration file with one option per line. You can modify it in a text editor, but it is preferable to use a specialized editor. On Linux, there is ccmake, which is ncurses-based (simple text-based GUI inside the terminal). On Windows, there is cmake-gui. These tools are very useful in order to visualize and edit the entire set of options available for the current project.

3.1.4 Configuration Files

If you have to frequently regenerate the CMake, the task of entering all the options that you want over and over again will be cumbersome. A flexible build system also means a large number of options, and the process of typing all of them every time is not only slow but also error prone. This is where storing build options in a configuration file becomes useful.

Luckily for us, CMake supports using values from an initial cache when generating a new one. The format of the initial cache is the same as the CMakeCache.txt file. The easiest way to use the initial cache is to create a text file and the desired options from a previously generated cache. Since many values in the cache are specific to the current build environment, it is not recommend to copy the entire generated cache to be used as an initial cache.

Here is an example: when developing, I like to keep a working version of FreeRDP installed in a directory separated from other less stable builds. Such a build comes in handy when I need to use FreeRDP rather than develop it, and my current development build is broken. For this build, I use the following options when invoking CMake:

cmake –DCMAKE_INSTALL_PREFIX=”/opt/freerdp” –DCMAKE_BUILD_TYPE=Release –DSTATIC_CHANNELS=on –DMONOLITHIC_BUILD=on –DWITH_SERVER=on .

In the resulting CMakeCache.txt file, the options I’ve passed will look like this:

WITH_SERVER:BOOL=on

STATIC_CHANNELS:BOOL=on

MONOLITHIC_BUILD:BOOL=on

CMAKE_BUILD_TYPE:STRING=Release

CMAKE_INSTALL_PREFIX:STRING=/opt/freerdp

You will notice that the format in the CMake cache is a bit different that the format used at the command-line. In reality, they turn out to be the same,

except that CMake guesses the parameter type when you invoke it. Values like "on/off" are recognized as BOOL, but if you were to explicitly specify the STRING data type, CMake would consider "on/off" as a STRING rather than a BOOL.

The above options can be copied in a text file which I will call "StableBuildCache.txt" for the sake of this example. The next time, instead of retyping everything, all that is needed is passing the initial cache to CMake using the –C option:

cmake –C StableBuildCache.txt .

This will have the same effect as passing all the desired parameters through the command-line, with much less effort.

3.1.5 Out-of-Source Build

One of the powerful features of CMake is the ability to make out-of-source builds. Normally, a build is made in-source, meaning that build scripts and build outputs are produced in the same directories as the sources. An out-of-source build, as the name says, produces build scripts and build outputs outside of the sources.

To generate out-of-source build scripts, invoke CMake from a directory different from the source tree, while providing the path to the directory where the root CMake script is contained. For instance, if I have my sources in ~ but I want to build everything inside the ~ directory, here is what the command would look like:

cmake ../../src/FreeRDP

Where the current working directory is ~

3.2 Options

CMake options are passed to the cmake command, prefixed with –D (D for Define). BOOL options are either ON or OFF. STRING options are just regular strings, and can be put between quotes.

3.2.1 CMake Options

CMAKE_BUILD_TYPE (STRING [Release]): Defines the build type, usually Debug or Release.

BUILD_SHARED_LIBS (BOOL [OFF]): Build libraries with an unspecified build type as shared libraries. This option is turned on by default in FreeRDP.

CMAKE_INSTALL_PREFIX (STRING): Path prefix for installation. On Linux, this is /usr/local by default.

3.2.2 Build Options

MONOLITHIC_BUILD (BOOL [OFF]): Combine smaller libraries into single, larger, monolithic libraries. In monolithic build mode, there is a single libfreerdp instead of multiple libraries such as libfreerdp-core, libfreerdp-cache, libfreerdp-utils, etc.

STATIC_CHANNELS (BOOL [ON]): Build all channels statically, including channels normally built as plugins. Client static channels are bundled in libfreerdp-client, and server static channels are bundled in libfreerdp-server.

BUILD_TESTING (BOOL [OFF]): Enable CTest and build unit tests. Unit tests are located in "test" subdirectories, and can be executed with the ctest command.

BUILD_SAMPLE (BOOL [OFF]): Build sample code, such as sample channels, clients or servers.

3.2.3 Optimization Options

WITH_SSE2 (BOOL): Build with SSE2 CPU instructions support (Intel architecture only).

WITH_NEON (BOOL): Build with NEON CPU instructions support (ARM architecture only).

3.2.4 Client Options

WITH_CLIENT (BOOL [ON]): Build clients

WITH_CLIENT_CHANNELS (BOOL [ON]): Build client channels.

3.2.5 Server Options

WITH_SERVER (BOOL [OFF]): Build servers

WITH_SERVER_CHANNELS (BOOL [OFF]): Build server channels.

3.2.6 Channel Options

WITH_CHANNELS (BOOL [ON]): Build channels.

Channel Names:

- AUDIN (Audio Input)
- CLIPRDR (Clipboard Redirection)
- DRIVE (Drive / File System Redirection)
- DRDYNVC (Dynamic Virtual Channel)
- PARALLEL (Parallel Port Redirection)
- PRINTER (Printer Redirection)
- RAIL (Remote Applications)
- RDPDR (Device Redirection)
- RDPSND (Audio Output)
- SERIAL (Serial Port Redirection)
- SMARTCARD (Smart Card Redirection)
- TSMF (Multimedia Redirection)
- URBDRC (USB Redirection)

CHANNEL_ (BOOL): Build channels

CHANNEL_NAME_CLIENT (BOOL): Build client channel. This option is dependent on CHANNEL_.

CHANNEL_NAME_SERVER (BOOL): Build server channel. This option is dependent on CHANNEL_.

3.2.7 Feature Options

WITH_JPEG: Build with JPEG codec support. This feature is not part of the canonical RDP specifications, and is implemented as an alternative codec to RemoteFX or NSCodec.

WITH_WIN8: Build with Windows 8 support. This is used by the Windows FreeRDP server and enables linking against Windows 8 libraries, therefore breaking backwards compatibility.

Chapter 4

Testing

4.1 Cunit

A cunit directory is present in the FreeRDP source tree. Building CUnit tests can be enabled with the WITH_CUNIT option. However, CUnit tests are deprecated, and current unit tests will eventually be migrated to CTest. New tests should NOT be written using CUnit.

4.1.1 CTest

CTest unit tests are located in "test" subdirectories and are deeply integrated with the rest of the CMake family of software process tools. CTest unit tests do not need a particular test framework. Test code is written as simple programs which return an exit code indicating success or failure.

CTest unit tests can be enabled with the BUILD_TESTING option.

Introduction

All unit tests can be executed by invoking ctest with no option:

ctest .

Unit tests can be filtered by name using a regular expression and the –R option.

To run all tests with a name beginning with "TestPath", use:

ctest –R "TestPath*" .

To execute only the "TestPathCchAppend" test, use:

ctest –R "TestPathCchAppend$" .

Debug output is turned off by default, but it can be enabled with the –V (verbose) option:

ctest –V –R “TestPathCchAppend$” .