how to install curl and libcurl

Installing Binary Packages

Lots of people download binary distributions of curl and libcurl. This document does not describe how to install curl or libcurl using such a binary package. This document describes how to compile, build and install curl and libcurl from source code.

Building using vcpkg

You can download and install curl and libcurl using the vcpkg dependency manager:

```
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh
./vcpkg integrate install
vcpkg install curl[tool]
```

The curl port in vcpkg is kept up to date by Microsoft team members and community contributors. If the version is out of date, please create an issue or pull request on the vcpkg repository.

Building from git

If you get your code off a git repository instead of a release tarball, see the `GIT-INFO` file in the root directory for specific instructions on how to proceed.

Unix

A normal Unix installation is made in three or four steps (after you've unpacked the source archive):

```
./configure
make
make test (optional)
make install
```

You probably need to be root when doing the last command.

Get a full listing of all available configure options by invoking it like:

```
./configure --help
```

If you want to install curl in a different file hierarchy than `/usr/local`, specify that when running configure:

```
./configure --prefix=/path/to/curl/tree
```

If you have write permission in that directory, you can do 'make install' without being root. An example of this would be to make a local install in your own home directory:

```
./configure --prefix=$HOME
make
make install
```

The configure script always tries to find a working SSL library unless explicitly told not to. If you have OpenSSL installed in the default search path for your compiler/linker, you don't need to do anything special. If you have OpenSSL installed in `/usr/local/ssl`, you can run configure like:

```
./configure --with-ssl
```

If you have OpenSSL installed somewhere else (for example, `/opt/OpenSSL`) and you have pkg-config installed, set the pkg-config path first, like this:

```
env PKG_CONFIG_PATH=/opt/OpenSSL/lib/pkgconfig ./configure --with
```

Without pkg-config installed, use this:

```
./configure --with-ssl=/opt/OpenSSL
```

If you insist on forcing a build without SSL support, even though you may have OpenSSL installed in your system, you can run configure like this:

```
./configure --without-ssl
```

If you have OpenSSL installed, but with the libraries in one place and the header files somewhere else, you have to set
the `LDFLAGS` and `CPPFLAGS` environment variables prior to running configure. Something like this should work:

```
CPPFLAGS="-I/path/to/ssl/include" LDFLAGS="-L/path/to/ssl/lib" ./
```

If you have shared SSL libs installed in a directory where your run-time linker doesn't find them (which usually causes configure failures), you can provide this option to gcc to set a hard-coded path to the run-time linker:

```
LDFLAGS=-Wl,-R/usr/local/ssl/lib ./configure --with-ssl
```

More Options

To force a static library compile, disable the shared library creation by running configure like:

```
./configure --disable-shared
```

To tell the configure script to skip searching for thread-safe functions, add an option like:

```
./configure --disable-thread
```

If you're a curl developer and use gcc, you might want to enable more debug options with the `--enable-debug` option.

curl can be built to use a whole range of libraries to provide various useful services, and configure will try to auto-detect a decent default. But if you want to alter it, you can select how to deal with each individual library.

Select TLS backend

The default OpenSSL configure check will also detect and use BoringSSL or libressl.

- GnuTLS: `--without-ssl --with-gnutls`.
- wolfSSL: `--without-ssl --with-wolfssl`
- NSS: `--without-ssl --with-nss`
- mbedTLS: `--without-ssl --with-mbedtls`
- schannel: `--without-ssl --with-schannel`
- secure transport: `--without-ssl --with-secure-transport`
- MesaLink: `--without-ssl --with-mesalink`
- BearSSL: `--without-ssl --with-bearssl`

Windows

Building Windows DLLs and C run-time (CRT) linkage issues

As a general rule, building a DLL with static CRT linkage is highly discouraged, and intermixing CRTs in the same app is something to avoid at any cost.

Reading and comprehending Microsoft Knowledge Base articles KB94248 and KB140584 is a must for any Windows developer. Especially important is full understanding if you are not going to follow the advice given above.

- How To Use the C Run-Time
- Run-Time Library Compiler Options
- Potential Errors Passing CRT Objects Across DLL Boundaries

If your app is misbehaving in some strange way, or it is suffering from memory corruption, before asking for further help, please try first to rebuild every single library your app uses as well as your app using the debug multithreaded dynamic C runtime.

If you get linkage errors read section 5.7 of the FAQ document.

MingW32

Make sure that MinGW32's bin dir is in the search path, for example:

```
set PATH=c:\mingw32\bin;%PATH%
```

then run `mingw32-make mingw32` in the root dir. There are other make targets available to build libcurl with more features, use:

- `mingw32-make mingw32-zlib` to build with Zlib support;
- `mingw32-make mingw32-ssl-zlib` to build with SSL and Zlib enabled;
- `mingw32-make mingw32-ssh2-ssl-zlib` to build with SSH2, SSL, Zlib;
- `mingw32-make mingw32-ssh2-ssl-sspi-zlib` to build with SSH2, SSL, Zlib and SSPI support.

If you have any problems linking libraries or finding header files, be sure to verify that the provided `Makefile.m32` files use the proper paths, and adjust as necessary. It is also possible to override these paths with environment variables, for example:

```
set ZLIB_PATH=c:\zlib-1.2.8
set OPENSSL_PATH=c:\openssl-1.0.2c
set LIBSSH2_PATH=c:\libssh2-1.6.0
```

It is also possible to build with other LDAP SDKs than MS LDAP; currently it is possible to build with native Win32 OpenLDAP, or with the Novell CLDAP SDK. If you want to use these you need to set these vars:

```
set LDAP_SDK=c:\openldap
set USE_LDAP_OPENLDAP=1
```

or for using the Novell SDK:

```
set USE_LDAP_NOVELL=1
```

If you want to enable LDAPS support then set LDAPS=1.

Cygwin

Almost identical to the unix installation. Run the configure script in the curl source tree root with `sh configure`. Make sure you have the `sh` executable in `/bin/` or you'll see the configure fail toward the end.

Run `make`

Disabling Specific Protocols in Windows builds

The configure utility, unfortunately, is not available for the Windows environment, therefore, you cannot use the various disable-protocol options of the configure utility on this platform.

You can use specific defines to disable specific protocols and features. SeeCURL-DISABLE.md for the full list.

If you want to set any of these defines you have the following options:

- Modify `lib/config-win32.h`
- Modify `lib/curl_setup.h`
- Modify `winbuild/Makefile.vc`
- Modify the "Preprocessor Definitions" in the libcurl project

Note: The pre-processor settings can be found using the Visual Studio IDE under "Project -> Settings -> C/C++ -> General" in VC6 and "Project -> Properties -> Configuration Properties -> C/C++ -> Preprocessor" in later versions.

Using BSD-style lwIP instead of Winsock TCP/IP stack in Win32 builds

In order to compile libcurl and curl using BSD-style lwIP TCP/IP stack it is necessary to make definition of preprocessor symbol `USE_LWIPSOCK` visible to libcurl and curl compilation processes. To set this definition you have the following alternatives:

- Modify `lib/config-win32.h` and `src/config-win32.h`
- Modify `winbuild/Makefile.vc`
- Modify the "Preprocessor Definitions" in the libcurl project

Note: The pre-processor settings can be found using the Visual Studio IDE under "Project -> Settings -> C/C++ -> General" in VC6 and "Project -> Properties -> Configuration Properties -> C/C++ -> Preprocessor" in later versions.

Once that libcurl has been built with BSD-style lwIP TCP/IP stack support, in order to use it with your program it is mandatory that your program includes lwIP header file `<lwip/opt.h>` (or another lwIP header that includes this) before including any libcurl header. Your program does not need the `USE_LWIPSOCK`preprocessor definition which is for libcurl internals only.

Compilation has been verified with lwIP 1.4.0 and contrib-1.4.0.

This BSD-style lwIP TCP/IP stack support must be considered experimental given that it has been verified that lwIP 1.4.0 still needs some polish, and libcurl might yet need some additional adjustment, caveat emptor.

Important static libcurl usage note

When building an application that uses the static libcurl library on Windows, you must add `-DCURL_STATICLIB` to your `CFLAGS`. Otherwise the linker will look for dynamic import symbols.

Legacy Windows and SSL

Schannel (from Windows SSPI), is the native SSL library in Windows. However, Schannel in Windows <= XP is unable to connect to servers that no longer support the legacy handshakes and algorithms used by those versions. If you will be using curl in one of those earlier versions of Windows you should choose another SSL backend such as OpenSSL.

Apple iOS and macOS

On modern Apple operating systems, curl can be built to use Apple's SSL/TLS implementation, Secure Transport, instead of OpenSSL. To build with Secure Transport for SSL/TLS, use the configure option `--with-secure-transport`. (It is not necessary to use the option `--without-ssl`.) This feature requires iOS 5.0 or later, or OS X 10.5 ("Leopard") or later.

When Secure Transport is in use, the curl options `--cacert` and `--capath` and their libcurl equivalents, will be ignored, because Secure Transport uses the certificates stored in the Keychain to evaluate whether or not to trust the server. This, of course, includes the root certificates that ship with the OS. The `--cert`and `--engine` options, and their libcurl equivalents, are currently unimplemented in curl with Secure Transport.

For macOS users: In OS X 10.8 ("Mountain Lion"), Apple made a major overhaul to the Secure Transport API that, among other things, added support for the newer TLS 1.1 and 1.2 protocols. To get curl to support TLS 1.1 and 1.2, you must build curl on Mountain Lion or later, or by using the equivalent SDK. If you set the `MACOSX_DEPLOYMENT_TARGET` environmental variable to an earlier version of macOS prior to building curl, then curl will use the new Secure Transport API on Mountain Lion and later, and fall back on the older API when the same curl binary is executed on older cats. For example, running these commands in curl's directory in the shell will build the code such that it will run on cats as old as OS X 10.6 ("Snow Leopard") (using bash):

```
export MACOSX_DEPLOYMENT_TARGET="10.6"
./configure --with-secure-transport
make
```

Android

When building curl for Android it's recommended to use a Linux environment since using curl's `configure` script is the easiest way to build curl for Android. Before you can build curl for Android, you need to install the Android NDK first. This can be done using the SDK Manager that is part of Android Studio. Once you have installed the Android NDK, you need to figure out where it has been installed and then set up some environment variables before launching `configure`. On macOS, those variables could look like this to compile for `aarch64` and API level 29:

```
export NDK=~/Library/Android/sdk/ndk/20.1.5948944
export HOST_TAG=darwin-x86_64
export TOOLCHAIN=$NDK/toolchains/llvm/prebuilt/$HOST_TAG
export AR=$TOOLCHAIN/bin/aarch64-linux-android-ar
export AS=$TOOLCHAIN/bin/aarch64-linux-android-as
export CC=$TOOLCHAIN/bin/aarch64-linux-android29-clang
export CXX=$TOOLCHAIN/bin/aarch64-linux-android29-clang++
export LD=$TOOLCHAIN/bin/aarch64-linux-android-ld
export RANLIB=$TOOLCHAIN/bin/aarch64-linux-android-ranlib
export STRIP=$TOOLCHAIN/bin/aarch64-linux-android-strip
```

When building on Linux or targeting other API levels or architectures, you need to adjust those variables accordingly. After that you can build curl like this:

```
./configure --host aarch64-linux-android --with-pic --disable-shar
```

Note that this won't give you SSL/TLS support. If you need SSL/TLS, you have to build curl against a SSL/TLS layer, e.g. OpenSSL, because it's impossible for curl to access Android's native SSL/TLS layer. To build curl for Android using OpenSSL, follow the OpenSSL build instructions and then install `libssl.a` and`libcrypto.a` to `$TOOLCHAIN/sysroot/usr/lib` and copy `include/openssl` to`$TOOLCHAIN/sysroot/usr/include`. Now you can build curl for Android using OpenSSL like this:

```
./configure --host aarch64-linux-android --with-pic --disable-shar
```

Note, however, that you must target at least Android M (API level 23) or `configure` won't be able to detect OpenSSL since `stderr` (and the like) weren't defined before Android M.

Cross compile

Download and unpack the curl package.

`cd` to the new directory. (e.g. `cd curl-7.12.3`)

Set environment variables to point to the cross-compile toolchain and call configure with any options you need. Be sure and specify the `--host` and `--build` parameters at configuration time. The following script is an example of cross-compiling for the IBM 405GP PowerPC processor using the toolchain from MonteVista for Hardhat Linux.

```
#! /bin/sh
export PATH=$PATH:/opt/hardhat/devkit/ppc/405/bin
export CPPFLAGS="-I/opt/hardhat/devkit/ppc/405/target/usr/include'
export AR=ppc_405-ar
export AS=ppc_405-as
export LD=ppc_405-ld
export RANLIB=ppc_405-ranlib
export CC=ppc_405-gcc
export NM=ppc_405-nm
./configure --target=powerpc-hardhat-linux
    --host=powerpc-hardhat-linux
    --build=i586-pc-linux-gnu
    --prefix=/opt/hardhat/devkit/ppc/405/target/usr/local
    --exec-prefix=/usr/local
```

You may also need to provide a parameter like `--with-random=/dev/urandom` to configure as it cannot detect the presence of a random number generating device for a target system. The `--prefix` parameter specifies where curl will be installed. If `configure` completes successfully, do `make` and `make install` as usual.

In some cases, you may be able to simplify the above commands to as little as:

```
./configure --host=ARCH-OS
```

REDUCING SIZE

There are a number of configure options that can be used to reduce the size of libcurl for embedded applications where binary size is an important factor. First, be sure to set the `CFLAGS` variable when configuring with any relevant compiler optimization flags to reduce the size of the binary. For gcc, this would mean at minimum the -Os option, and potentially the `-march=X`, `-mdynamic-no-pic` and `-flto` options as well, e.g.

```
./configure CFLAGS='-Os' LDFLAGS='-Wl,-Bsymbolic'...
```

Note that newer compilers often produce smaller code than older versions due to improved optimization.

Be sure to specify as many `--disable-` and `--without-` flags on the configure command-line as you can to disable all the libcurl features that you know your application is not going to need. Besides specifying the `--disable-PROTOCOL`flags for all the types of URLs your application will not use, here are some other flags that can reduce the size of the library:

- `--disable-ares` (disables support for the C-ARES DNS library)
- `--disable-cookies` (disables support for HTTP cookies)
- `--disable-crypto-auth` (disables HTTP cryptographic authentication)
- `--disable-ipv6` (disables support for IPv6)
- `--disable-manual` (disables support for the built-in documentation)
- `--disable-proxy` (disables support for HTTP and SOCKS proxies)
- `--disable-unix-sockets` (disables support for UNIX sockets)
- `--disable-verbose` (eliminates debugging strings and error code strings)
- `--disable-versioned-symbols` (disables support for versioned symbols)
- `--enable-hidden-symbols` (eliminates unneeded symbols in the shared library)
- `--without-libidn` (disables support for the libidn DNS library)
- `--without-librtmp` (disables support for RTMP)
- `--without-ssl` (disables support for SSL/TLS)
- `--without-zlib` (disables support for on-the-fly decompression)

The GNU compiler and linker have a number of options that can reduce the size of the libcurl dynamic libraries on some platforms even further. Specify them by providing appropriate `CFLAGS` and `LDFLAGS` variables on the configure command-line, e.g.

```
CFLAGS="-Os -ffunction-sections -fdata-sections
        -fno-unwind-tables -fno-asynchronous-unwind-tables -flto"
LDFLAGS="-Wl,-s -Wl,-Bsymbolic -Wl,--gc-sections"
```

Be sure also to strip debugging symbols from your binaries after compiling using 'strip' (or the appropriate variant if cross-compiling). If space is really tight, you may be able to remove some unneeded sections of the shared library using the -R option to objcopy (e.g. the .comment section).

Using these techniques it is possible to create a basic HTTP-only shared libcurl library for i386 Linux platforms that is only 113 KiB in size, and an FTP-only library that is 113 KiB in size (as of libcurl version 7.50.3, using gcc 5.4.0).

You may find that statically linking libcurl to your application will result in a lower total size than dynamically linking.

Note that the curl test harness can detect the use of some, but not all, of the `--disable` statements suggested above. Use will cause tests relying on those features to fail. The test harness can be manually forced to skip the relevant tests by specifying certain key words on the `runtests.pl` command line. Following is a list of appropriate key words:

- `--disable-cookies` !cookies
- `--disable-manual` !--manual
- `--disable-proxy` !HTTP\ proxy !proxytunnel !SOCKS4 !SOCKS5

PORTS

This is a probably incomplete list of known hardware and operating systems that curl has been compiled for. If you know a system curl compiles and runs on, that isn't listed, please let us know!

- Alpha DEC OSF 4
- Alpha Digital UNIX v3.2
- Alpha FreeBSD 4.1, 4.5
- Alpha Linux 2.2, 2.4
- Alpha NetBSD 1.5.2
- Alpha OpenBSD 3.0
- Alpha OpenVMS V7.1-1H2
- Alpha Tru64 v5.0 5.1
- AVR32 Linux
- ARM Android 1.5, 2.1, 2.3, 3.2, 4.x
- ARM INTEGRITY
- ARM iOS
- Cell Linux
- Cell Cell OS
- HP-PA HP-UX 9.X 10.X 11.X
- HP-PA Linux
- HP3000 MPE/iX
- MicroBlaze uClinux
- MIPS IRIX 6.2, 6.5
- MIPS Linux
- OS/400
- Pocket PC/Win CE 3.0
- Power AIX 3.2.5, 4.2, 4.3.1, 4.3.2, 5.1, 5.2
- PowerPC Darwin 1.0
- PowerPC INTEGRITY
- PowerPC Linux
- PowerPC Mac OS 9
- PowerPC Mac OS X
- SH4 Linux 2.6.X
- SH4 OS21

- SINIX-Z v5
- Sparc Linux
- Sparc Solaris 2.4, 2.5, 2.5.1, 2.6, 7, 8, 9, 10
- Sparc SunOS 4.1.X
- StrongARM (and other ARM) RISC OS 3.1, 4.02
- StrongARM/ARM7/ARM9 Linux 2.4, 2.6
- StrongARM NetBSD 1.4.1
- Symbian OS (P.I.P.S.) 9.x
- TPF
- Ultrix 4.3a
- UNICOS 9.0
- i386 BeOS
- i386 DOS
- i386 eCos 1.3.1
- i386 Esix 4.1
- i386 FreeBSD
- i386 HURD
- i386 Haiku OS
- i386 Linux 1.3, 2.0, 2.2, 2.3, 2.4, 2.6
- i386 Mac OS X
- i386 MINIX 3.1
- i386 NetBSD
- i386 Novell NetWare
- i386 OS/2
- i386 OpenBSD
- i386 QNX 6
- i386 SCO unix
- i386 Solaris 2.7
- i386 Windows 95, 98, ME, NT, 2000, XP, 2003
- i486 ncr-sysv4.3.03 (NCR MP-RAS)
- ia64 Linux 2.3.99
- m68k AmigaOS 3
- m68k Linux
- m68k uClinux
- m68k OpenBSD
- m88k dg-dgux5.4R3.00
- s390 Linux
- x86_64 Linux
- XScale/PXA250 Linux 2.4
- Nios II uClinux