Frequently Asked Questions about ZLIB1.DLL

This document describes the design, the rationale, and the usage

of the official DLL build of zlib, named ZLIB1.DLL. If you have

general questions about zlib, you should see the file "FAQ" found

in the zlib distribution, or at the following location:

http://www.gzip.org/zlib/zlib\_faq.html

1. What is ZLIB1.DLL, and how can I get it?

- ZLIB1.DLL is the official build of zlib as a DLL.

(Please remark the character '1' in the name.)

Pointers to a precompiled ZLIB1.DLL can be found in the zlib

web site at:

http://www.zlib.net/

Applications that link to ZLIB1.DLL can rely on the following

specification:

\* The exported symbols are exclusively defined in the source

files "zlib.h" and "zlib.def", found in an official zlib

source distribution.

\* The symbols are exported by name, not by ordinal.

\* The exported names are undecorated.

\* The calling convention of functions is "C" (CDECL).

\* The ZLIB1.DLL binary is linked to MSVCRT.DLL.

The archive in which ZLIB1.DLL is bundled contains compiled

test programs that must run with a valid build of ZLIB1.DLL.

It is recommended to download the prebuilt DLL from the zlib

web site, instead of building it yourself, to avoid potential

incompatibilities that could be introduced by your compiler

and build settings. If you do build the DLL yourself, please

make sure that it complies with all the above requirements,

and it runs with the precompiled test programs, bundled with

the original ZLIB1.DLL distribution.

If, for any reason, you need to build an incompatible DLL,

please use a different file name.

2. Why did you change the name of the DLL to ZLIB1.DLL?

What happened to the old ZLIB.DLL?

- The old ZLIB.DLL, built from zlib-1.1.4 or earlier, required

compilation settings that were incompatible to those used by

a static build. The DLL settings were supposed to be enabled

by defining the macro ZLIB\_DLL, before including "zlib.h".

Incorrect handling of this macro was silently accepted at

build time, resulting in two major problems:

\* ZLIB\_DLL was missing from the old makefile. When building

the DLL, not all people added it to the build options. In

consequence, incompatible incarnations of ZLIB.DLL started

to circulate around the net.

\* When switching from using the static library to using the

DLL, applications had to define the ZLIB\_DLL macro and

to recompile all the sources that contained calls to zlib

functions. Failure to do so resulted in creating binaries

that were unable to run with the official ZLIB.DLL build.

The only possible solution that we could foresee was to make

a binary-incompatible change in the DLL interface, in order to

remove the dependency on the ZLIB\_DLL macro, and to release

the new DLL under a different name.

We chose the name ZLIB1.DLL, where '1' indicates the major

zlib version number. We hope that we will not have to break

the binary compatibility again, at least not as long as the

zlib-1.x series will last.

There is still a ZLIB\_DLL macro, that can trigger a more

efficient build and use of the DLL, but compatibility no

longer dependents on it.

3. Can I build ZLIB.DLL from the new zlib sources, and replace

an old ZLIB.DLL, that was built from zlib-1.1.4 or earlier?

- In principle, you can do it by assigning calling convention

keywords to the macros ZEXPORT and ZEXPORTVA. In practice,

it depends on what you mean by "an old ZLIB.DLL", because the

old DLL exists in several mutually-incompatible versions.

You have to find out first what kind of calling convention is

being used in your particular ZLIB.DLL build, and to use the

same one in the new build. If you don't know what this is all

about, you might be better off if you would just leave the old

DLL intact.

4. Can I compile my application using the new zlib interface, and

link it to an old ZLIB.DLL, that was built from zlib-1.1.4 or

earlier?

- The official answer is "no"; the real answer depends again on

what kind of ZLIB.DLL you have. Even if you are lucky, this

course of action is unreliable.

If you rebuild your application and you intend to use a newer

version of zlib (post- 1.1.4), it is strongly recommended to

link it to the new ZLIB1.DLL.

5. Why are the zlib symbols exported by name, and not by ordinal?

- Although exporting symbols by ordinal is a little faster, it

is risky. Any single glitch in the maintenance or use of the

DEF file that contains the ordinals can result in incompatible

builds and frustrating crashes. Simply put, the benefits of

exporting symbols by ordinal do not justify the risks.

Technically, it should be possible to maintain ordinals in

the DEF file, and still export the symbols by name. Ordinals

exist in every DLL, and even if the dynamic linking performed

at the DLL startup is searching for names, ordinals serve as

hints, for a faster name lookup. However, if the DEF file

contains ordinals, the Microsoft linker automatically builds

an implib that will cause the executables linked to it to use

those ordinals, and not the names. It is interesting to

notice that the GNU linker for Win32 does not suffer from this

problem.

It is possible to avoid the DEF file if the exported symbols

are accompanied by a "\_\_declspec(dllexport)" attribute in the

source files. You can do this in zlib by predefining the

ZLIB\_DLL macro.

6. I see that the ZLIB1.DLL functions use the "C" (CDECL) calling

convention. Why not use the STDCALL convention?

STDCALL is the standard convention in Win32, and I need it in

my Visual Basic project!

(For readability, we use CDECL to refer to the convention

triggered by the "\_\_cdecl" keyword, STDCALL to refer to

the convention triggered by "\_\_stdcall", and FASTCALL to

refer to the convention triggered by "\_\_fastcall".)

- Most of the native Windows API functions (without varargs) use

indeed the WINAPI convention (which translates to STDCALL in

Win32), but the standard C functions use CDECL. If a user

application is intrinsically tied to the Windows API (e.g.

it calls native Windows API functions such as CreateFile()),

sometimes it makes sense to decorate its own functions with

WINAPI. But if ANSI C or POSIX portability is a goal (e.g.

it calls standard C functions such as fopen()), it is not a

sound decision to request the inclusion of <windows.h>, or to

use non-ANSI constructs, for the sole purpose to make the user

functions STDCALL-able.

The functionality offered by zlib is not in the category of

"Windows functionality", but is more like "C functionality".

Technically, STDCALL is not bad; in fact, it is slightly

faster than CDECL, and it works with variable-argument

functions, just like CDECL. It is unfortunate that, in spite

of using STDCALL in the Windows API, it is not the default

convention used by the C compilers that run under Windows.

The roots of the problem reside deep inside the unsafety of

the K&R-style function prototypes, where the argument types

are not specified; but that is another story for another day.

The remaining fact is that CDECL is the default convention.

Even if an explicit convention is hard-coded into the function

prototypes inside C headers, problems may appear. The

necessity to expose the convention in users' callbacks is one

of these problems.

The calling convention issues are also important when using

zlib in other programming languages. Some of them, like Ada

(GNAT) and Fortran (GNU G77), have C bindings implemented

initially on Unix, and relying on the C calling convention.

On the other hand, the pre- .NET versions of Microsoft Visual

Basic require STDCALL, while Borland Delphi prefers, although

it does not require, FASTCALL.

In fairness to all possible uses of zlib outside the C

programming language, we choose the default "C" convention.

Anyone interested in different bindings or conventions is

encouraged to maintain specialized projects. The "contrib/"

directory from the zlib distribution already holds a couple

of foreign bindings, such as Ada, C++, and Delphi.

7. I need a DLL for my Visual Basic project. What can I do?

- Define the ZLIB\_WINAPI macro before including "zlib.h", when

building both the DLL and the user application (except that

you don't need to define anything when using the DLL in Visual

Basic). The ZLIB\_WINAPI macro will switch on the WINAPI

(STDCALL) convention. The name of this DLL must be different

than the official ZLIB1.DLL.

Gilles Vollant has contributed a build named ZLIBWAPI.DLL,

with the ZLIB\_WINAPI macro turned on, and with the minizip

functionality built in. For more information, please read

the notes inside "contrib/vstudio/readme.txt", found in the

zlib distribution.

8. I need to use zlib in my Microsoft .NET project. What can I

do?

- Henrik Ravn has contributed a .NET wrapper around zlib. Look

into contrib/dotzlib/, inside the zlib distribution.

9. If my application uses ZLIB1.DLL, should I link it to

MSVCRT.DLL? Why?

- It is not required, but it is recommended to link your

application to MSVCRT.DLL, if it uses ZLIB1.DLL.

The executables (.EXE, .DLL, etc.) that are involved in the

same process and are using the C run-time library (i.e. they

are calling standard C functions), must link to the same

library. There are several libraries in the Win32 system:

CRTDLL.DLL, MSVCRT.DLL, the static C libraries, etc.

Since ZLIB1.DLL is linked to MSVCRT.DLL, the executables that

depend on it should also be linked to MSVCRT.DLL.

10. Why are you saying that ZLIB1.DLL and my application should

be linked to the same C run-time (CRT) library? I linked my

application and my DLLs to different C libraries (e.g. my

application to a static library, and my DLLs to MSVCRT.DLL),

and everything works fine.

- If a user library invokes only pure Win32 API (accessible via

<windows.h> and the related headers), its DLL build will work

in any context. But if this library invokes standard C API,

things get more complicated.

There is a single Win32 library in a Win32 system. Every

function in this library resides in a single DLL module, that

is safe to call from anywhere. On the other hand, there are

multiple versions of the C library, and each of them has its

own separate internal state. Standalone executables and user

DLLs that call standard C functions must link to a C run-time

(CRT) library, be it static or shared (DLL). Intermixing

occurs when an executable (not necessarily standalone) and a

DLL are linked to different CRTs, and both are running in the

same process.

Intermixing multiple CRTs is possible, as long as their

internal states are kept intact. The Microsoft Knowledge Base

articles KB94248 "HOWTO: Use the C Run-Time" and KB140584

"HOWTO: Link with the Correct C Run-Time (CRT) Library"

mention the potential problems raised by intermixing.

If intermixing works for you, it's because your application

and DLLs are avoiding the corruption of each of the CRTs'

internal states, maybe by careful design, or maybe by fortune.

Also note that linking ZLIB1.DLL to non-Microsoft CRTs, such

as those provided by Borland, raises similar problems.

11. Why are you linking ZLIB1.DLL to MSVCRT.DLL?

- MSVCRT.DLL exists on every Windows 95 with a new service pack

installed, or with Microsoft Internet Explorer 4 or later, and

on all other Windows 4.x or later (Windows 98, Windows NT 4,

or later). It is freely distributable; if not present in the

system, it can be downloaded from Microsoft or from other

software provider for free.

The fact that MSVCRT.DLL does not exist on a virgin Windows 95

is not so problematic. Windows 95 is scarcely found nowadays,

Microsoft ended its support a long time ago, and many recent

applications from various vendors, including Microsoft, do not

even run on it. Furthermore, no serious user should run

Windows 95 without a proper update installed.

12. Why are you not linking ZLIB1.DLL to

<<my favorite C run-time library>> ?

- We considered and abandoned the following alternatives:

\* Linking ZLIB1.DLL to a static C library (LIBC.LIB, or

LIBCMT.LIB) is not a good option. People are using the DLL

mainly to save disk space. If you are linking your program

to a static C library, you may as well consider linking zlib

in statically, too.

\* Linking ZLIB1.DLL to CRTDLL.DLL looks appealing, because

CRTDLL.DLL is present on every Win32 installation.

Unfortunately, it has a series of problems: it does not

work properly with Microsoft's C++ libraries, it does not

provide support for 64-bit file offsets, (and so on...),

and Microsoft discontinued its support a long time ago.

\* Linking ZLIB1.DLL to MSVCR70.DLL or MSVCR71.DLL, supplied

with the Microsoft .NET platform, and Visual C++ 7.0/7.1,

raises problems related to the status of ZLIB1.DLL as a

system component. According to the Microsoft Knowledge Base

article KB326922 "INFO: Redistribution of the Shared C

Runtime Component in Visual C++ .NET", MSVCR70.DLL and

MSVCR71.DLL are not supposed to function as system DLLs,

because they may clash with MSVCRT.DLL. Instead, the

application's installer is supposed to put these DLLs

(if needed) in the application's private directory.

If ZLIB1.DLL depends on a non-system runtime, it cannot

function as a redistributable system component.

\* Linking ZLIB1.DLL to non-Microsoft runtimes, such as

Borland's, or Cygwin's, raises problems related to the

reliable presence of these runtimes on Win32 systems.

It's easier to let the DLL build of zlib up to the people

who distribute these runtimes, and who may proceed as

explained in the answer to Question 14.

13. If ZLIB1.DLL cannot be linked to MSVCR70.DLL or MSVCR71.DLL,

how can I build/use ZLIB1.DLL in Microsoft Visual C++ 7.0

(Visual Studio .NET) or newer?

- Due to the problems explained in the Microsoft Knowledge Base

article KB326922 (see the previous answer), the C runtime that

comes with the VC7 environment is no longer considered a

system component. That is, it should not be assumed that this

runtime exists, or may be installed in a system directory.

Since ZLIB1.DLL is supposed to be a system component, it may

not depend on a non-system component.

In order to link ZLIB1.DLL and your application to MSVCRT.DLL

in VC7, you need the library of Visual C++ 6.0 or older. If

you don't have this library at hand, it's probably best not to

use ZLIB1.DLL.

We are hoping that, in the future, Microsoft will provide a

way to build applications linked to a proper system runtime,

from the Visual C++ environment. Until then, you have a

couple of alternatives, such as linking zlib in statically.

If your application requires dynamic linking, you may proceed

as explained in the answer to Question 14.

14. I need to link my own DLL build to a CRT different than

MSVCRT.DLL. What can I do?

- Feel free to rebuild the DLL from the zlib sources, and link

it the way you want. You should, however, clearly state that

your build is unofficial. You should give it a different file

name, and/or install it in a private directory that can be

accessed by your application only, and is not visible to the

others (i.e. it's neither in the PATH, nor in the SYSTEM or

SYSTEM32 directories). Otherwise, your build may clash with

applications that link to the official build.

For example, in Cygwin, zlib is linked to the Cygwin runtime

CYGWIN1.DLL, and it is distributed under the name CYGZ.DLL.

15. May I include additional pieces of code that I find useful,

link them in ZLIB1.DLL, and export them?

- No. A legitimate build of ZLIB1.DLL must not include code

that does not originate from the official zlib source code.

But you can make your own private DLL build, under a different

file name, as suggested in the previous answer.

For example, zlib is a part of the VCL library, distributed

with Borland Delphi and C++ Builder. The DLL build of VCL

is a redistributable file, named VCLxx.DLL.

16. May I remove some functionality out of ZLIB1.DLL, by enabling

macros like NO\_GZCOMPRESS or NO\_GZIP at compile time?

- No. A legitimate build of ZLIB1.DLL must provide the complete

zlib functionality, as implemented in the official zlib source

code. But you can make your own private DLL build, under a

different file name, as suggested in the previous answer.

17. I made my own ZLIB1.DLL build. Can I test it for compliance?

- We prefer that you download the official DLL from the zlib

web site. If you need something peculiar from this DLL, you

can send your suggestion to the zlib mailing list.

However, in case you do rebuild the DLL yourself, you can run

it with the test programs found in the DLL distribution.

Running these test programs is not a guarantee of compliance,

but a failure can imply a detected problem.

\*\*

This document is written and maintained by

Cosmin Truta <cosmint@cs.ubbcluj.ro>