The following paper was originally published in the
Proceedings of the 2nd USENIX Windows NT Symposium
Seattle, Washington, August 3–4, 1998

Cygwin32: A Free Win32 Porting Layer for UNIX Applications

Geoffrey J. Noer
Cygnus Solutions

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

Cygwin32: A Free Win32 Porting Layer for UNIX® Applications

Geoffrey J. Noer
noer@cygnus.com

Cygnus Solutions
1325 Chesapeake Terrace
Sunnyvale, CA 94089

Abstract

Cygwin32 is a full-featured Win32 porting layer for UNIX applications, compatible with all Win32 hosts (currently Microsoft Windows NT, Windows 95, and Windows 98). It was invented in 1995 by Cygnus Solutions as part of the answer to the question of how to port the GNU development tools to the Win32 host.

The Win32-hosted GNUPro compiler tools that use the library are available for a variety of embedded processors as well as a native version for writing Win32 programs. By basing this technology on the GNU tools, Cygnus provides developers with a high-performance, feature-rich 32-bit code development environment, including a graphical source-level debugger.

Cygwin32 is a Dynamic-Linked Library (DLL) that provides a large subset of the system calls found in common UNIX implementations. The current release includes all POSIX.1/90 calls except for `setuid` and `mkfifo`, all ANSI C standard calls, and many common BSD and SVR4 services including Berkeley sockets.

This article will discuss our experiences porting the GNU development tools to the Win32 host and explore the development and architecture of the Cygwin32 library.

1. Introduction

Cygnus Solutions was founded in 1989 to provide commercial support and development services for the GNU development tools, focusing on the embedded computing industry. The tools include the GNU C/C++ compiler (GCC/G++), assembler (GAS), linker (GLD), and debugger (GDB). As of June 1998, Cygnus sells support for over 150 host/target combinations.

When the Free Software Foundation (FSF) first wrote the GNU tools in the mid-1980s, portability among existing and future UNIX operating systems was an important goal. By mid-1995, the tools had been ported to 16-bit DOS using the go32 32-bit extender by D.J. Delorie[1]. However, no one had completed a native 32-bit port for Windows NT and 95/98. It seemed likely that the demand for Win32-hosted native and cross-development tools would soon be large enough to justify the development costs involved.

2. Porting the GNU Compiler to Win32

The first step in porting the compiler tools to Win32 was to enhance them so that they could generate and interpret Win32 native object files, using Microsoft's Portable Executable (PE) format. This proved to be relatively straightforward because of similarities to the Common Object File Format (COFF), which the GNU tools already supported. Most of these changes were confined to the Binary File Descriptor (BFD) library and to the linker.

In order to support the Win32 Application Programming Interface (API), we extended the capabilities of the binary utilities to handle Dynamic-Linked Libraries (DLLs). After creating export lists for the specific Win32 API DLLs that are shipped with Win32 hosts, the tools were able to generate static libraries that executables could use to gain access to Win32 API functions. Because of redistribution restrictions on Microsoft's Win32 API header files, we wrote our own Win32 header files from scratch on an as-needed basis. Once this work was completed, we were able to build UNIX-hosted cross-compilers capable of generating valid PE executables that ran on Win32 systems.

The next task was to port the compiler tools themselves to Win32. Previous experiences using Microsoft Visual C++ to port GDB convinced us to find another means for bootstrapping the full set of tools. In addition to

wanting to use our own compiler technology, we wanted a portable build system. The GNU development tools' configuration and build procedures require a large number of additional UNIX utilities not available on Win32 hosts. So we decided to use UNIX-hosted cross-compilers to build our Win32-hosted native and cross-development tools. It made perfect sense to do this since we were successfully using a nearly identical technique to build our DOS-hosted products.

The next obstacle to overcome was the many dependencies on UNIX system calls in the sources, especially in the GNU debugger GDB. While we could have rewritten sizable portions of the source code to work within the context of the Win32 API (as was done for the DOS-hosted tools), this would have been prohibitively time-consuming. Worse, we would have introduced conditionalized code that would have been expensive to maintain in the long run. Instead, Cygnus developers took a substantially different approach by writing Cygwin32.

3. Initial Goals

The original goal of Cygwin32 was simply to get the development tools working. Completeness with respect to POSIX.1[2] and other relevant UNIX standards was not a priority.

Part of our definition of "working native tools" is having a build environment similar enough to UNIX to support rebuilding the tools themselves on the host system, a process we call self-hosting. The typical configuration procedure for a GNU tool involves running "configure", a complex Bourne shell script that determines information about the host system. The script then uses that information to generate the Makefiles used to build the tool on the host in question.

This configuration mechanism is needed under UNIX because of the large number of varying flavors of UNIX. If Microsoft continues to produce new variants of the Win32 API as it releases new versions of its operating systems, it may prove to be quite valuable on the Win32 host as well.

The need to support this configuration procedure added the requirement of supporting user tools such as sh, make, file utilities (e.g. ls and rm), text utilities (e.g. cat, tr), shell utilities (e.g. echo, date, uname), sed, awk, find, xargs, tar, and gzip, among many others. Previously, most of these user tools had only been built natively (on the host on which they would run). As a result, we had to modify their configure scripts to be compatible with cross-compilation.

Other than making the necessary configuration changes, we wanted to avoid Win32-specific changes since the UNIX compatibility was to be provided by Cygwin32 as much as possible. While we knew this would be a sizable amount of work, there was more to gain than just achieving self-hosting of the tools. Supporting the configuration of the development tools would also provide an excellent method of testing the Cygwin32 library.

Although we were able to build working Win32-hosted toolchains with cross-compilers relatively soon after the birth of Cygwin32, it took much longer than we expected before the tools could reliably rebuild themselves on the Win32 host because of the many complexities involved.

4. "Harnessing the Power of the Internet"

Instead of keeping the Cygwin32 technology proprietary and developing it in-house, Cygnus chose to make it publicly available under the terms of the GNU General Public License (GPL), the traditional license for the GNU tools. Since its inception, we have made a new "GNU-Win32 beta release" available via ftp over the Internet every three or four months. Each release includes binaries of Cygwin32 and the development tools, coupled with the source code needed to build them. Unlike standard Cygnus products, these free releases come without any assurances of quality or support, although we provide a mailing list that is used for discussion and feedback.

In retrospect, making the technology freely available was a good decision because of the high demand for quality 32-bit native tools in the Win32 arena, as well as significant additional interest in a UNIX portability layer like Cygwin32. While far from perfect, the beta releases are good enough for many people. They provide us with tens of thousands of interested developers who are willing to use and test the tools. A few of them are even willing to contribute code fixes and new functionality to the library. As of the last public release, developers on the Net had written or improved a significant portion of the library, including important aspects such as support for UNIX signals and the TTY/PTY calls.

In order to spur as much Net participation as possible, the Cygwin32 project features an open development model. We make weekly source snapshots available to the general public in addition to the periodic full GNU-

Win32 releases. A developers' mailing list facilitates discussion of proposed changes to the library.

In addition to the GPL version of Cygwin32, Cygnus provides a commercial license for supported customers of the native Win32 GNUPro tools.

5. The Cygwin32 Architecture

Now we turn to an analysis of the actual architecture of the Cygwin32 library.

When a binary linked against the library is executed, the Cygwin32 DLL is loaded into the application's text segment. Because we are trying to emulate a UNIX kernel which needs access to all processes running under it, the first Cygwin32 DLL to run creates shared memory areas that other processes using separate instances of the DLL can access. This is used to keep track of open file descriptors and assist `fork` and `exec`, among other purposes. In addition to the shared memory regions, every process also has a `per_process` structure that contains information such as process id, user id, signal masks, and other similar process-specific information.

The DLL is implemented using the Win32 API, which allows it to run on all Win32 hosts. Because processes run under the standard Win32 subsystem, they can access both the UNIX compatibility calls provided by Cygwin32 as well as any of the Win32 API calls. This gives the programmer complete flexibility in designing the structure of their program in terms of the APIs used. For example, they could write a Win32-specific GUI using Win32 API calls on top of a UNIX back-end that uses Cygwin32.

Early on in the development process, we made the important design decision that it would not be necessary to strictly adhere to existing UNIX standards like POSIX.1 if it was not possible or if it would significantly diminish the usability of the tools on the Win32 platform. In many cases, an environment variable can be set to override the default behavior and force standards compliance.

5.1. Windows NT != Windows 95/98

While Windows 95 and Windows 98 are similar enough to each other that we can safely ignore the distinction when implementing Cygwin32, Windows NT is an extremely different operating system. For this reason, whenever the DLL is loaded, the library checks which operating system is active so that it can act accordingly.

In some cases, the Win32 API is only different for historical reasons. In this situation, the same basic functionality is available under 95/98 and NT but the method used to gain this functionality differs. A trivial example: in our implementation of `uname`, the library examines the `sysinfo.dwProcessorType` structure member to figure out the processor type under 95/98. This field is not supported in NT, which has its own operating system-specific structure member called `sysinfo.wProcessorLevel`.

Other differences between NT and 95/98 are much more fundamental in nature. The best example is that only NT provides a security model.

5.2. Permissions and Security

Windows NT includes a sophisticated security model based on Access Control Lists (ACLs). Although some modern UNIX operating systems include support for ACLs, Cygwin32 maps Win32 file ownership and permissions to the more standard, older UNIX model. The `chmod` call maps UNIX-style permissions back to the Win32 equivalents. Because many programs expect to be able to find the `/etc/passwd` and `/etc/group` files, we provide utilities that can be used to construct them from the user and group information provided by the operating system.

Under Windows NT, the administrator is permitted to `chown` files. There is currently no mechanism to support the `setuid` concept or API call. Although we hope to support this functionality at some point in the future, in practice, the programs we have ported have not needed it.

Under Windows 95/98, the situation is considerably different. Since a security model is not provided, Cygwin32 fakes file ownership by making all files look like they are owned by a default user and group id. As under NT, file permissions can still be determined by examining their read/write/execute status. Rather than return an unimplemented error, under Windows 95/98, the `chown` call succeeds immediately without actually performing any action whatsoever. This is appropriate since essentially all users jointly own the files when no concept of file ownership exists.

It is important that we discuss the implications of our "kernel" using shared memory areas to store information about Cygwin32 processes. Because these areas are not yet protected in any way, in principle a malicious user could modify them to cause unexpected behavior in Cygwin32 processes. While this is not a new problem under Windows 95/98 (because of the

lack of operating system security), it does constitute a security hole under Windows NT. This is because one user could affect the Cygwin32 programs run by another user by changing the shared memory information in ways that they could not in a more typical WinNT program. For this reason, it is not appropriate to use Cygwin32 in high-security applications. In practice, this will not be a major problem for most uses of the library.

5.3. Files

Cygwin32 supports both Win32- and POSIX-style paths, using either forward or back slashes as the directory delimiter. Paths coming into the DLL are translated from Win32 to POSIX as needed. As a result, the library believes that the file system is a POSIX-compliant one, translating paths back to Win32 paths whenever it calls a Win32 API function. UNC pathnames (starting with two slashes) are supported.

The layout of this POSIX view of the Windows file system space is stored in the Windows registry. While the slash ('/') directory points to the system partition by default, this is easy to change with the Cygwin32 `mount` utility. In addition to selecting the slash partition, it allows mounting arbitrary Win32 paths into the POSIX file system space. Many people use the utility to mount each drive letter under the slash partition (e.g. `C:\` to `/c`, `D:\` to `/d`, etc…).

The library exports several Cygwin32-specific functions that can be used by external programs to convert a path or path list from Win32 to POSIX or vice versa. Shell scripts and Makefiles cannot call these functions directly. Instead, they can do the same path translations by executing the "cygpath" utility program that we provide with Cygwin32.

Win32 file systems are case preserving but case insensitive. Cygwin32 does not currently support case distinction because, in practice, few UNIX programs actually rely on it. While we could mangle file names to support case distinction, this would add unnecessary overhead to the library and make it more difficult for non-Cygwin32 applications to access those files.

Symbolic links are emulated by files containing a magic cookie followed by the path to which the link points. They are marked with the System attribute so that only files with that attribute have to be read to determine whether or not the file is a symbolic link. Hard links are fully supported under Windows NT on NTFS file systems. On a FAT file system, the call falls back to simply copying the file, a strategy that works in many cases.

The inode number for a file is calculated by hashing its full Win32 path. The inode number generated by the `stat` call always matches the one returned in `d_ino` of the `dirent` structure. It is worth noting that the number produced by this method is not guaranteed to be unique. However, we have not found this to be a significant problem because of the low probability of generating a duplicate inode number.

5.4. Text Mode vs. Binary Mode

Interoperability with other Win32 programs such as text editors was critical to the success of the port of the development tools. Most Cygnus customers upgrading from the older DOS-hosted toolchains expected the new Win32-hosted ones to continue to work with their old development sources.

Unfortunately, UNIX and Win32 use different end-of-line terminators in text files. Consequently, carriage-return newlines have to be translated on the fly by Cygwin32 into a single newline when reading in text mode. The control-z character is interpreted as a valid end-of-file character for a similar reason.

This solution addresses the compatibility requirement at the expense of violating the POSIX standard that states that text and binary mode will be identical. Consequently, processes that attempt to `lseek` through text files can no longer rely on the number of bytes read as an accurate indicator of position in the file. For this reason, an environment variable can be set to override this behavior.

5.5. ANSI C Library

We chose to include Cygnus' own existing ANSI C[3] library "newlib" as part of the library, rather than write all of the lib C and math calls from scratch. Newlib is a BSD-derived ANSI C library, previously only used by cross-compilers for embedded systems development.

The reuse of existing free implementations of such things as the `glob`, `regexp`, and `getopt` libraries saved us considerable effort. In addition, Cygwin32 uses Doug Lea's free `malloc` implementation that successfully balances speed and compactness. The library accesses the `malloc` calls via an exported function pointer. This makes it possible for a Cygwin32 process to provide its own malloc if it so desires.

5.6. Process Creation

The `fork` call in Cygwin32 is particularly interesting because it does not map well on top of the Win32 API. This makes it very difficult to implement correctly. Currently, the Cygwin32 `fork` is a non-copy-on-write implementation similar to what was present in early flavors of UNIX.

The first thing that happens when a parent process forks a child process is that the parent initializes a space in the Cygwin32 process table for the child. It then creates a suspended child process using the Win32 `CreateProcess` call. Next, the parent process calls `setjmp` to save its own context and sets a pointer to this in a Cygwin32 shared memory area (shared among all Cygwin32 tasks). It then fills in the child's `.data` and `.bss` sections by copying from its own address space into the suspended child's address space. After the child's address space is initialized, the child is run while the parent waits on a mutex. The child discovers it has been forked and longjumps using the saved jump buffer. The child then sets the mutex the parent is waiting on and blocks on another mutex. This is the signal for the parent to copy its stack and heap into the child, after which it releases the mutex the child is waiting on and returns from the `fork` call. Finally, the child wakes from blocking on the last mutex, recreates any memory-mapped areas passed to it via the shared area, and returns from `fork` itself.

While we have some ideas as to how to speed up our `fork` implementation by reducing the number of context switches between the parent and child process, `fork` will almost certainly always be inefficient under Win32. Fortunately, in most circumstances the `spawn` family of calls provided by Cygwin32 can be substituted for a `fork`/`exec` pair with only a little effort. These calls map cleanly on top of the Win32 API. As a result, they are much more efficient. Changing the compiler's driver program to call `spawn` instead of `fork` was a trivial change and increased compilation speeds by twenty to thirty percent in our tests.

However, `spawn` and `exec` present their own set of difficulties. Because there is no way to do an actual `exec` under Win32, Cygwin32 has to invent its own Process IDs (PIDs). As a result, when a process performs multiple `exec` calls, there will be multiple Windows PIDs associated with a single Cygwin32 PID. In some cases, stubs of each of these Win32 processes may linger, waiting for their `exec'd` Cygwin32 process to exit.

5.7. Signals

When a Cygwin32 process starts, the library starts a secondary thread for use in signal handling. This thread waits for Windows events used to pass signals to the process. When a process notices it has a signal, it scans its signal bitmask and handles the signal in the appropriate fashion.

Several complications in the implementation arise from the fact that the signal handler operates in the same address space as the executing program. The immediate consequence is that Cygwin32 system functions are interruptible unless special care is taken to avoid this. We go to some lengths to prevent the `sig_send` function that sends signals from being interrupted. In the case of a process sending a signal to another process, we place a mutex around `sig_send` such that `sig_send` will not be interrupted until it has completely finished sending the signal.

In the case of a process sending itself a signal, we use a separate semaphore/event pair instead of the mutex. `sig_send` starts by resetting the event and incrementing the semaphore that flags the signal handler to process the signal. After the signal is processed, the signal handler signals the event that it is done. This process keeps intraprocess signals synchronous, as required by POSIX.

Most standard UNIX signals are provided. Job control works as expected in shells that support it.

5.8. Sockets

Socket-related calls in Cygwin32 simply call the functions by the same name in Winsock, Microsoft's implementation of Berkeley sockets. Only a few changes were needed to match the expected UNIX semantics — one of the most troublesome differences was that Winsock must be initialized before the first socket function is called. As a result, Cygwin32 has to perform this initialization when appropriate. In order to support sockets across `fork` calls, child processes initialize Winsock if any inherited file descriptor is a socket.

Unfortunately, implicitly loading DLLs at process startup is usually a slow affair. Because many processes do not use sockets, Cygwin32 explicitly loads the Winsock DLL the first time it calls the Winsock initialization routine. This single change sped up GNU configure times by thirty percent.

5.9. Select

The UNIX `select` function is another call that does not map cleanly on top of the Win32 API. Much to our dismay, we discovered that the Win32 `select` in Winsock only worked on socket handles. Our implementation allows `select` to function normally when given different types of file descriptors (sockets, pipes, handles, and a custom `/dev/windows` windows messages pseudo-device).

Upon entry into the `select` function, the first operation is to sort the file descriptors into the different types. There are then two cases to consider. The simple case is when at least one file descriptor is a type that is always known to be ready (such as a disk file). In that case, `select` returns immediately as soon as it has polled each of the other types to see if they are ready. The more complex case involves waiting for socket or pipe file descriptors to be ready. This is accomplished by the main thread suspending itself, after starting one thread for each type of file descriptor present. Each thread polls the file descriptors of its respective type with the appropriate Win32 API call. As soon as a thread identifies a ready descriptor, that thread signals the main thread to wake up. This case is now the same as the first one since we know at least one descriptor is ready. So `select` returns, after polling all of the file descriptors one last time.

6. Performance

Early on in the development process, correctness was almost the entire emphasis. As Cygwin32 became more complete, performance became a much important issue. We knew that the tools ran much more slowly under Win32 than under Linux on the same machine, but it was not clear at all whether to attribute this to differences in the operating systems or to inefficiencies in Cygwin32.

The lack of a working profiler has made analyzing Cygwin32's performance particularly difficult. Although the latest version of the library includes "real" itimer support, we have not yet found a way to implement virtual itimers. This is the most reliable way of obtaining profiling data since concurrently running processes aren't likely to skew the results. We will soon have a combination of the gcc compiler and the GNU profile analysis tool gprof working with "real" itimer support which will help a great deal in optimizing Cygwin32.

Even without a profiler, we knew of several areas inside Cygwin32 that definitely needed a fresh approach. While we rewrote those sections of code, we used the speed of configuring the tools under Win32 as the primary performance measurement. This choice made sense because we knew process creation speed was especially poor, something that the GNU configure process stresses.

These performance adjustments made it possible to completely configure the development tools under NT with Cygwin32 in only ten minutes and complete the build in just under an hour on a dual Pentium Pro 200 system with 128 MB of RAM. This is reasonably competitive with the time taken to complete this task under a typical flavor of the UNIX operating system running on an identical machine.

7. Ported Software

In addition to being able to configure and build most GNU software, several other significant packages have been successfully ported to the Win32 host using the Cygwin32 library. Following is a list of some of the more interesting ones (most are not included in the free Internet distributions):

- X11R6 client libraries, enabling porting many X programs to the existing free Win32 X servers. Examples of successfully ported X applications include xterm, ghostview, xfig, and xconq.
- xemacs and vim editors.
- GNU inetutils. It is possible to run the inetd daemon as a Windows NT service to enable UNIX-style networking, using a custom NT login binary to allow remote logins with full user authentication. One can achieve similar results under Windows 95/98 by running inetd out of the `autoexec.bat` file, providing a custom 95/98-tailored login binary.
- KerbNet, Cygnus' implementation of the kerberos security system.
- CVS (Concurrent Versions System), a popular version control program based on RCS. Cygnus uses a kerberos-enabled version of CVS to grant secure access to our source code to local and remote engineers.
- ncurses, a library that can be used to build a functioning version of the pager "less".
- ssh (secure shell) client and server.

- PERL 5 scripting language.
- The bash, tcsh, ash, and zsh shells. Full job control is available in shells that support it.
- Apache web server (some source-level changes were necessary).
- TCL/TK 8; also tix, itcl, and expect. (TCL/TK needed non-trivial configuration changes).

Typically, the only necessary source code modification involves specifying binary mode to `open` calls as appropriate. Because our Win32 compiler always generates executables that end in the standard `.exe` suffix, it is also often necessary to make minor modifications to makefiles so that make will expect the newly built executables to end with the suffix.

8. Future Work

Standards conformance is becoming a more important focus. In the last release, all POSIX.1/90 calls are implemented except for `mkfifo` and `setuid`. X/Open Release 4[4] conformance may be a desirable goal, but we have not pursued this yet. While the current version of the library passes most of the NIST POSIX test suite[5] with flying colors, it performs poorly with respect to mimicking the UNIX security model, so there is still room for improvement. When we consider how to implement the `setuid` functionality, we will also look into a secure alternative to the library's usage of the shared memory areas.

Cygwin32 does not yet support applications that use multiple Windows threads, even though the library itself is multi-threaded. We expect to address this shortcoming through the use of locks at strategic points in the DLL. It would also be desirable to implement support for POSIX threads.

Although Cygwin32 allows the GNU development tools that depend heavily on UNIX semantics to successfully run on Win32 hosts, it is not always desirable to use it. A program using a perfect implementation of the library would still incur a noticeable amount of overhead. As a result, an important future direction involves modifying the compiler so that it can optionally link against the Microsoft DLLs that ship with both Win32 operating systems, instead of Cygwin32. This will give developers the ability to choose whether or not to use Cygwin32 on a per-program basis.

9. Proprietary Alternatives

When we started developing Cygwin32, alternatives to writing our own library either did not exist or were not mature enough for our purposes. Today, we know of three proprietary alternatives to Cygwin32: UWIN from AT&T Laboratories, NuTCracker from DataFocus, and OpenNT from Softway Systems.

UWIN[6] ("UNIX for Windows") was developed by David Korn for AT&T Laboratories. Its architecture and API appears to be quite similar to our library. Its single biggest advantage over Cygwin32 is probably its more complete support for the UNIX security model. UWIN binaries are available for free non-commercial use, but its source code is not available.

NuTCracker, by DataFocus, is another proprietary product that is built on top of the Win32 subsystem. Version 4.0 of the product appears to be quite complete, including such features as support for POSIX threads.

OpenNT from Softway Systems[7] takes a markedly different approach by providing a capable POSIX subsystem for Windows NT, implemented with the Windows NT source code close at hand. At least in principle, writing a separate POSIX subsystem should result in better performance because of the lack of overhead imposed when implementing a library on top of the Win32 subsystem. More importantly, by avoiding the compromises inherent in supporting both Win32 and POSIX calls in one application, it should be possible for OpenNT to conform more strictly to the relevant standards.

However, there are two substantial drawbacks to OpenNT's approach. The first is that it is not possible to mix UNIX and Win32 API calls in one application, a feature that is highly desirable if you are attempting to do a full native Win32 port of a UNIX program gradually, one module at a time. The second drawback is that OpenNT does not and cannot support Windows 95/98, a requirement for many applications, including the GNUPro development tools.

The lack of source code, coupled with the licensing fees associated with each of these commercial offerings, might still have required us to have written our own library if we were faced with the same porting challenge today.

10. Summary and Conclusions

Cygwin32 is a UNIX-compatibility library that can be used to port UNIX software to Win32 operating sys-

tems. In this paper, I have examined our motivations for writing Cygwin32. I have analyzed its architecture in some detail, paying extra attention to those areas where UNIX and Win32 differ the most. I have listed examples of successfully ported software and touched on performance issues. I have discussed where we expect to take Cygwin32 in the future. Finally, I have described the proprietary alternatives to our library.

As you can see from the list of ported software presented earlier in this paper, Cygwin32 can be used to facilitate greatly the process of porting significant UNIX applications to Win32 hosts. For some applications, it may be desirable to invest in a true native Win32 port in order to remove the overhead imposed by Cygwin32. However, the increased portability and time saved by using Cygwin32 should make it an attractive option in many situations.

11. Availability

Please consult our project WWW page to obtain more information about Cygwin32, including how to download the latest source code and binary release:

http://www.cygnus.com/misc/gnu-win32

For more information about the GNUPro development tools, please visit:

http://www.cygnus.com/product/gnupro.html

12. Acknowledgements

The author wishes to thank the many other people who have helped write Cygwin32, in particular Steve Chamberlain who wrote the original implementation of the library. Jeremy Allison, Doug Evans, Christopher Faylor, Philippe Giacinti, Tim Newsham, Sergey Okhapkin, and Ian Taylor have all made significant contributions to the library. The author also appreciates the feedback and proofreading help given to him by Eric Bachalo, Chip Chapin, Christopher Faylor, Kathleen Jones, Robert Richardson, Stan Shebs, Sonya Smallets, and Ethan Solomita, as well as from Stephan Walli, his USENIX paper advisor.

13. Trademarks

---

[1] D.J. Delorie: The DJGPP Project. Available from http://www.delorie.com/djgpp.

[2] ISO/IEC 9945-1:1996. (ANSI/IEEE Std 1003.1, 1996 Edition) — POSIX Part 1: System Application Program Interface (API) [C Language].

[3] ISO/IEC 9899:1990, Programming Languages — C.

[4] The X/Open Release 4 CAE Specification, System Interfaces and Headers, Issue 4, Vol. 2, X/Open Co, Ltd., 1994.

[5] NIST POSIX test suite. Available from http://www.itl.nist.gov/div897/ctg/posix_form.htm.

[6] Korn, David G. UWIN — UNIX for Windows. Proceedings of the 1997 USENIX Windows NT Annual Technical Conference.

[7] Walli, Stephen R. OpenNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem. Proceedings of the 1997 USENIX Windows NT Workshop Proceedings.