A Critique of the Windows Application Programming Interface *†

Diomidis Spinellis
University of the Aegean
83200 Karlovasi
Greece
email: dspin@aegean.gr

December 1997

Abstract

The architecture, interface, and functionality of the Windows Application Programming Interface (API) make it difficult to master and use effectively, and contribute negatively to the safety, robustness, and portability of the applications developed under it. The API is structured around a large and constantly evolving set of functions and is based on a problematic shared library implementation. The provided interfaces are complicated, non-orthogonal, abuse the type system, cause name-space pollution, and use inconsistent naming conventions. In addition, the functionality of the interface suffers from inconsistency, incompleteness, and inadequate documentation. Application developers, programming tool vendors, and Microsoft should face the above problems and provide appropriate solutions.

Keywords: *Microsoft Windows; Application Programming Interface; Win32*

1 Introduction

Microsoft Windows 95 and Windows NT (from now on referred-to as "Windows") are increasingly becoming widely adopted as operating system platforms for desktop applications, back-office servers, and research [1]. Their programming interface, currently distributed and documented as the "Microsoft Platform Software Development Kit" [2] (SDK), provides a set of functions, data types, structures, macros, and tools for writing user and system software to run under Windows. Although application writers can be isolated from the SDK by using libraries, scripting, visual and fourth-generation languages, or utilising programmable components, ultimately the SDK provides the operating system interface thus affecting the robustness, portability, performance, safety, and ease of Windows programming.

The first versions of Windows provided a graphical environment to the MS-DOS operating system. The current versions of Windows provide a 32-bit graphical, multi-tasking, networked operating system [3] used by thousands of workstation and server applications. The core SDK Application Programming Interface (API) covers an extremely broad area providing the interfaces listed bellow.

Input and output devices: mouse, keyboard, pen, screen, printer, and sound.

User interface elements: windows, menus, dialogs, input widgets, the clipboard, and internationalisation functions.

System services: files, memory, hardware, system databases, and networking.

Graphical elements: bitmaps, fonts, drawing primitives, area management functions), and 3D graphics rendering.

An additional number of APIs are provided and documented as part of the Windows Platform SDK. The use of some of them is required in order to develop an application that will satisfy the licensing requirements of Microsoft's "Designed for Windows NT and Windows 95" Logo Program. These APIs cover the following areas:

- the Microsoft's Component Object Model (COM), Object Linking and Embedding (OLE), application automation, and ActiveX,
- shell interfacing,

Computer Standards & Interfaces, 20:1–8, November 1998.

†This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work.

- telephony interfaces (TAPI),
- remote access services and procedure calls (RPC),
- Internet networking, W3 server interfacing (Winsock, ISAPI),
- messaging (MAPI), and
- game applications (DirectX 2).

The Windows platform SDK also documents a number of interfaces for entities that are not yet part of the standard Windows distributions such as the Microsoft SQL, Transaction, and Exchange servers, the management console and clustering interfaces, the Win32 Internet functions, and the Open Database Connectivity Interface (ODBC). Although many of the shortcomings of the basic Windows API are also evident in the above mentioned interfaces, we will not cover these in this article.

The Windows interface is specified using C language bindings, although due to the nature of its implementation — as a set of shared libraries callable using the calling convention commonly associated with Pascal programs — many of its functions are accessible from other languages and programming environments.

In this article we will critically examine the architecture, interface, and functionality of the Windows API and point to a number of problems associated with it. We will argue that because of these problems the Windows interface:

- is difficult to master and use effectively,
- can be used to distort competition in the marketplace,
- contributes negatively to the safety, robustness, and portability of the applications developed under it.

The remainder of this article is structured as follows: in the next section we examine the API's *structure*, size, and implementation looking on how these affect software development, reliability, and marketplace competition. In section 3 we examine the *interface* provided by the Windows API and identify problems related to the complexity and non-orthogonality of the provided interfaces, the type system, name-space pollution, inconsistent naming conventions, and portability. In section 4 we look beyond the interface into the actual *functionality* provided by the API and provide examples of inconsistency, inadequate documentation, and incompleteness. Finally, the last section contains proposals on how application developers, programming tool vendors, and Microsoft should handle the identified API problems.

Element	Number
Number of root header files	129
Number of import libraries	48
Total number of header files	232
Header file size (Mb)	5.2
Header file lines (non empty non comment)	120516
Macro and constant definitions	33174
Type definitions	4858
Functions	3433
Interface methods	1462
Messages	858
Notification messages	180
Structures	1077
Properties	498
Enumeration types	110
Function error codes	1137

Table 1: Win32 API key metrics

2 Size, Structure, and Implementation

The Windows API is accessed through a very large and complicated set of elements. Its size is difficult to judge because what exactly constitutes it is far from clear. The Windows SDK definition and its contents change rapidly according to Microsoft's strategic and marketing interests. As an example the October 1996 edition of the Microsoft Development Library documents the Internet Server API (ISAPI) as part of the Win32 Software Development Kit (SDK), but documents other server related APIs (such as the Open Database Connectivity — ODBC — API) as separate entities. The April 1997 version of the Microsoft Development Library documents all Windows interfaces under the roof of a single "Platform SDK".

In this article we will consider the Windows API (*Win32*) to consist of the items supplied as parts of Microsoft's Win32 Software Development Kit. The POSIX subsystem of Windows NT, although part of the Win32 SDK, is separately installed and documented; for this reason we will not consider it as part of Win32.

A file (WIN32API.CSV) supplied together with the Win32 SDK lists 9067 API elements (functions, interface methods, structures, messages, macros, properties, etc.) This number although large does not include about 29000 constant definitions (constants defined using the #define mechanism of the C preprocessor) and about 4800 type definitions (C typedefs) that can be found by going through all C header files that are part of the SDK, nor does it include the Unicode, ASCII, and character set neutral function forms. A summary of some key metric sizes of the Win32 API is provided in Table 1.

The large size and monolithic nature of the Win32 API negatively affect a number of areas related to software development. The huge number elements comprising the API make it difficult to master it and use it effectively. As a result the productivity of application architects, software developers, and maintainers is negatively affected.

In addition, the creation of systems providing the same services on different platforms is difficult, and, given the rapidly evolving nature of the API, could well be impossible. In the past, major advances in research and development of new hardware and operating system architectures such as the RISC processors and microkernels were leveraged on the ability to provide a Unix-like environment on top of the new architecture. With the domination of the Windows API new hardware and software architectures, in order to be accepted, will need to support the Windows API. Microsoft's exclusive control of the API can distort competition and market diversity.

Finally, given the size of the API, any formal proof of specific properties or the correctness of programs using it is an extremely difficult task. As a result, either the reliability of life-critical software will suffer, or such software will be developed, maintained, and operated in an environment isolated from the rest of the mainstream software. This will have important cost and interoperability consequences.

Apart from its large size, one other problem related to the API structure is its reliance on a shared library system, the Windows Dynamic Linked Libraries (DLLs). The current implementation of the API and its binding mechanism provide no version and interfacing control over the applications that use DLLs and the libraries they are linked to. Although DLLs can be associated with a version number, at a given time only a single version of a DLL can be loaded on the system. As a result major library interface changes, such as the transition to 32 bit code, rely on a haphazard mixture of simple renaming and replacing of library modules for satisfying the new linkage requirements. One exemplar result of this simple-minded approach is that application installation disks created with the Visual Basic 3.0 development environment on a Windows 95 platform will destroy the setup of a Windows 3.1 platform when an installation is attempted. In other cases where compatibility with older software had to be preserved, as was the case with the introduction of the Jet 2.0 database engine, a complicated set of new library modules and stubs had to be correctly installed for the system to function.

In addition to the above, the monolithic structure and large size of the API contribute to name-space pollution problems that it creates. Any non-trivial Windows application will need to include the *windows.h* header file which in turn includes more than 60 other header files comprising more than 70000 lines of C declarations and macro definitions.

3 Interface

The provided functions have a complex and non-intuitive interface with a number of mode changing flags and exceptions that unnecessarily complicate system application development. Space restrictions do not allow us to provide a detailed example; interested readers are encouriaged to discover for their own edification the three different ways in which a read-only mode can be specified using the seven parameters of the *CreateFile* function.

Despite the apparent generality of functions such as *CreateFile* it would be a mistake to think that the Windows API provides a small set of generalised functions that cover a lot of ground by being combined in an orthogonal fashion. Win32 provides 91 functions that create entities (from *CreateAcceleratorTable* to *CreateWindowStation*. All those functions receive parameters of different types in wildly differing order; even similar functions that provide enhanced functionality (such as *CopyFileEx*) have the new arguments interspersed with the existing ones. The return value of the functions that create entities is also inconsistent. The following are representative examples of return value inconsistencies across functions that create different entities:

CreatePipe returns TRUE for success and FALSE on error.

CreateFile returns a handle to the file object on success and the INVALID_HANDLE_VALUE constant on error.

CreateFileMapping returns a handle to the mapping object on success and NULL on error.

CreateTapePartition returns NO_ERROR on success and one of 15 constants (ERROR_BEGINNING_OF_MEDIA to ERROR_WRITE_PROTECT) on error.

CreateHalftonePalette returns a handle to the palette object on success and *zero* on error.

The complexity of the API increases even more with the provision of 131 "extended" functions (ending in *Ex*) that perform similar tasks to the original ones, but provide extended or sometimes just different functionality. For example, the *CreateWindowEx* function provides an additional parameter for specifying 21 "extended" window styles in addition to the 139 styles (27 basic and 112 class-dependent) allowed by the *CreateWindow* function, while *WriteFileEx* provides the functionality of *WriteFile*, but is designed solely for asynchronous operation. In addition to the above, 1226 functions exist in *three flavours* according to the character set they support: Unicode, ANSI (an 8-bit superset of the ASCII character set), and character set neutral. The Unicode and ANSI versions of the functions are named by appending the letter "U", or "A" respectively after the function name. The character set neutral functions are defined as a C preprocessor macro that calls one of the other two functions depending on the source code compilation specifications.

3.1 Type System Problems

Although the current specification of ANSI C provides a type system that can be used to detect many type errors at compile time, the Windows API specification provides ample opportunities to break it by specifying in a large number of cases arguments with minimal type information associated with them.

Older releases of Windows declared the various "handles" (small integer constants used for identifying operating system entities) in a way that made them type compatible. Thus it was possible to pass to an API function that expected a window handle, a handle to a device context or a handle to a brush. The situation has improved with later releases of the Windows API which can (with the definition of the C preprocessor's "STRICT" symbol) perform type checking across different types of entity handles.

Other type-related problems persist. More than 150 functions pass an argument of type LPVOID or PVOID) which is a pointer to any type, in effect short-circuiting the compiler's type checking system. Some of the functions (e.g. *CopyMemory*) use this argument type legitimately for providing an interface to unstructured memory. Other functions however, typically pass a pointer of an appropriate type depending on the value of another argument. As an example the *GetTokenInformation* function which is used to retrieve a specified type of information about an access token can pass as an argument a pointer to ten different structures (TOKEN_USER to TOKEN_STATISTICS depending on the class of the requested token information which is also specified as an argument.

An even worse abuse of the type system is performed through the use of the LPARAM and WPARAM types. These simply specify 32 bit and 16 bit values respectively which are used (after suitable casting and without any type checks) for any purpose. The 32 bit value is often used to pass flags, integer values, pointers to memory, pointers to functions, or even packed data combined into 32 bits. In the current version of the Windows API 89 functions have an LPARAM argument and 48 a WPARAM argument. These types are also used for declaring structure members circumventing the type system in one additional way. The worst offender in this category is probably the MSG structure which contains message information. The structure contains both an LPARAM and a WPARAM which are used for different purposes for each one of the 180 notification messages. When the two structure members are not enough for passing all the message related data LPARAM is simply used to pass a pointer to a structure containing additional information.

One last problem with the way the API types are defined stems from the extensive use of the WORD and DWORD types. These are defined as 16 bit and 32 bit unsigned values and are used for passing flags and integer values. Their definition in effect guarantees an API implementation detail limiting the API's portability to future architectures with a different natural word size.

3.2 Namespace Pollution

As mentioned in section 2, the large size of the Windows API increases the namespace pollution it creates. As the C language provides only three types of visibility (function, file, and global) all the API functions, types, and constants have to be defined with global visibility. This means that their names are exposed and can interfere with other functions that the end-user application defines. The Windows API names have no unique prefix (as do the X-Window system API names) thus compounding the problem. Even if an application does not use any of the thousands of names used by the API there is no guarantee that a new version of the API will not use a name also used by the application. The provision of separate namespaces for structure tags and enumeration constants by the C language does not significantly help the namespace pollution problem, because constants and macros defined using the C preprocessor mechanism can potentially interfere with any other type of name. In addition, the way C modules and libraries are usually linked virtually guarantees that if an application defines a global function with the same name as a Windows API function, the application's function will simply replace the API's function at link time without even a warning.

3.3 Function Names and Naming Conventions

The Windows API naming conventions are inconsistent making the functions and constants hard to remember. This problem is somehow mitigated by the excellent help available through the full text searchable hypertext pages provided my Microsoft, but is nevertheless annoying.

The capitalisation of acronyms is performed inconsistently. For example, the functions belonging to the *audio video interleaved* (AVI) function group are named prefixed with an uppercase AVI (.e.g *AVIFileOpen*) whereas most functions belonging to the *media control interface* (MCI) function group are prefixed with a lowercase *mci* (e.g. *mciSendCommand*), and the functions belonging to the *remote access service* (RAS) group are prefixed with "Ras" (e.g. *RasAdminFreeBuffer*). In addition, although the splitting of words within a function name is mostly performed by capitalising the first character of every word (with the first letter of the first word not capitalised when it is used to specify a function group) a number of functions split the words using a combination of capitalisation and underscores (e.g. *ImageList_Add*). Furthermore, some functions that are inherited from other APIs such as the socket or string handling functions have their names written using lowercase characters.

Many function names consist of a group name, a verb,

and an object. Unfortunately no particular order is used when forming the above items into a function name. Often the group to which a function belongs is prefixing the function name as is the case in the AVI, MCI, and RAS function groups showed in the previous paragraph. In other cases, functions that could be grouped together, start off with a verb as is the case for functions used for manipulating drawing brushes: *CreateSolidBrush*, *FixBrushOrgEx*, *GetBrushOrgEx*, and *SetBrushOrgEx*. Some of the functions are formed with the verb followed by the object (e.g. *ReadConsole*, *ReadEventLog*), while others are formed by an object followed by a verb (e.g. *BackupRead*, *NetErrorLogRead*).

Furthermore, a given task is not always accurately reflected by the name of the function that accomplishes it. For example, in order to determine a disk volume's sector size the *GetDiskFreeSpaceEx* function has to be called.

3.4 Portability

The interface provided by the Windows API does not follow any established API standards such as POSIX. It is therefore difficult to port existing system-programming applications directly to it without resorting to the use of a relatively *thick* compatibility layer that isolates the application from the Windows system. Two such layers exist today: the Windows NT POSIX subsystem, and the GNU Win 32 project [4]. Both allow programs that are based on POSIX services to compile and run, but the resulting applications are isolated from the rest of the system in an "emulated" environment. Furthermore, the large size and pervasive nature of the Windows API makes it difficult to write applications that use the API and can still be ported to other environments. The difficult porting experience of Microsoft's Word program to the Macintosh platform proves this point [5, p. 150–151].

In addition, important portability problems exist even across the Windows systems that support the API. Of the 3433 available API functions, 45 are not supported under Windows NT, 602 are not supported under Windows-95 (26 of them were supported after the OSR-2 release), and 2125 are not supported under *Win32s* (a set of drivers and library files for Windows 3.1 that upgrade them to provide a part of the 32 bit Windows API functionality).

The above numbers reflect functions supported by the *current* versions of Windows systems. The API continuously changes in important ways as new versions or even service releases of Windows are brought out; as a result older versions of Windows do not support the new elements. A list of elements that were introduced across versions is presented in Table 2. It is important to note that *some* of the newer API elements can be added to older versions of Windows by installing the specific components together with the product that uses them.

Windows Release	New API elements
Windows NT 3.5	488
Windows 95	1168
Windows NT 3.51	581
Windows NT 3.51 Service Pack 3	6
Windows NT 4.0	500
Windows NT 4.0 Service Pack 2	6
Windows NT 4.0 Service Pack 3	6
Windows NT 5.0 (proposed)	27

Table 2: Additions to the API over successive Windows releases

4 Functionality

The functionality provided by the Windows API is in a number of cases inconsistent, inadequately documented, or incomplete. These shortcomings directly affect the reliability of applications developed under it. A particular example of inconsistent behaviour is the accessibility checking of memory addresses passed to functions. Some functions check that the addresses are accessible (e.g. *ReadProcessMemory*) and return an error if they are not, while others (e.g. *CopyMemory*, *GetSystemInfo*) do not perform any such checking and will cause the calling program to fail with a "general protection fault" if a bad address is passed to the function.

Error handling is a particular cause of problems. Some API functions clear the thread's global error code variable when they succeed while others do not. The documentation of most functions does not completely specify which of the 1130 errors can occur in a call to that function making it therefore impossible to anticipate them and recover in a sensible manner. Another API functionality problem stems from the incomplete documentation of the semantics of many functions. As a particular example, the *CreateProcess* function does not completely specify which of the current process attributes are inherited by the new process.

Some functions provide extremely rudimentary, low-level functionality imposing to the application an implementation cost that should have been covered by the operating system. The handling of asynchronous input and output operations provides a characteristic example. After an asynchronous write operation is initiated, any modifications of its buffer can corrupt the data being written. This design decision obviously avoids the buffer copy overhead, but imposes to the application the cost of a complicated dirty buffer management scheme. Even worse, if the implementor does not notice this trap, the end result will be a difficult to reproduce, non-deterministic application bug. The operating system could easily provide more sophisticated functionality using internally a "copy on write" buffer

management scheme.

An example of incomplete API functionality can be found in the video capture API. Although it is possible to set the video source using a dialog box displayed to the user by calling the *capDlgVideoSource* function, doing the same from within an application without user mediation is, as far as we know, impossible.

The Windows API is based on an event processing model. Applications have to continuously process events posted by the operating system in order to exhibit the requisite liveness properties and be compliant with the provided interface. This event model clashes with a number of application designs, algorithms, legacy applications, and system requirement specifications. Although the event processing model is pervasive in the Windows API, it is not unique. Other models for asynchronous operation have to be used in parallel, including *callbacks* (application-specified functions that are called by the operating system at a specific times), and a generic blocking facility provided by the *WaitFor* family of functions. The combination of these synchronisation primitives with the API's threads facility provides fertile ground for programming situations that can lead to a deadlock. Some examplar situations that can result in an application or even a system deadlock are the following:

- sending a message to a thread that yields control after receiving it, either directly or by calling a dialog box or a *GetMessage* function,
- calling the Windows API while holding a level 3 (device) lock,
- creating windows and calling the *Sleep* function with an infinite delay,
- calling *wave* (waveform audio services) output functions from within a *wave* callback procedure, and,
- failing to acknowledge an incoming dynamic data exchange DDE request.

5 Conclusions

In the previous sections we have described some shortcomings of the Windows API. Due to the widespread deployment of Windows-based systems it is important to recognise the problems associated with the API and deal with them at the appropriate level.

End-user application designers and developers can shield themselves from the direct use of the API through the use of high level and domain specific languages and libraries that isolate the application developer from the API specifics. System programmers could try to isolate the API specific part of their application or add an intermediate layer to it in order to enhance the application's portability and maintainability.

Programming language, library, and tool vendors should try to provide well-designed, generic, high-level functionality for accomplishing Windows related tasks, resisting the current worrying trend and natural temptation to cover the interfacing area by a cover-all Windows API gateway function.

However, the most important contribution to the Windows API long-term viability must ultimately come from the company that controls its future. Microsoft should recognise its responsibility in the market place and — breaking away with the past — invest effort in the design of a high-level, orthogonal, intuitive, structured, complete, and extendable API that will cover its own and the industry's current and future needs. Backwards compatibility with the current API could be provided with the development of mapping layer libraries. The inevitable performance cost of this change [6] can be absorbed by a single processor generation. Current experience shows that increases in processor power are delivered as increasingly sophisticated advances in graphical user interfaces (GUIs). Investing one generation of processor power increase in an architectural overhaul of the API will result in a payback in increased programmer productivity, program robustness, and portability worth more than the currently diminishing returns of GUI improvements.

References

[1] Usenix Association. *USENIX Windows NT Workshop*, Seattle, Washington, USA, August 1997.

[2] Microsoft Corporation. Microsoft platform software development kit. Distributed through the Microsoft Developer's Network Library, July 1997.

[3] Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, USA, 1992.

[4] Cygnus support. The GNU Win-32 project page. http://www.cygnus.com/misc/gnu-win32/.

[5] Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets*. The Free Press, 1995.

[6] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazières, Antonio Dias, Margo Seltzer, and Michael D. Smith. The measured performance of personal computer operating systems. *ACM Transactions on Computer Systems*, 14(1):3–40, February 1996.