

Arnold McDonald

L'APINATIVE

Version 1.01 – 1^{er} octobre 2002



QUACK

<http://perso.club-internet.fr/amcd/quack>

La plus grande attention a été apportée à la réalisation de ce document afin de présenter les informations les plus fiables et exactes possibles. Néanmoins, Quack et l'auteur n'assument aucune garantie, explicite ou implicite, concernant ces informations. Elles sont fournies « telles quelles » et l'utilisateur assume seul l'entière responsabilité des risques ou dommages éventuels pouvant résulter de leur utilisation.

Les codes source éventuellement présents ont pour seul but d'illustrer les propos de l'auteur. Ils ne sont nullement destinés à une utilisation commerciale ou professionnelle. Quack et l'auteur ne peuvent être, en aucune manière, tenus pour responsables des préjudices ou dommages, de quelque nature que ce soit, pouvant découler de leur emploi.

Quack et l'auteur n'endossent aucune responsabilité quant aux contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de l'utilisation des informations ou des codes sources de ce document.

Tous les noms de produits ou marques cités peuvent être des marques déposées par leurs propriétaires respectifs.

Correction : Pocky6.

Photo de couverture : Sandrine Bascouert.

Copyright © 2002, Quack
<http://perso.club-internet.fr/amcd/quack>.

Tous droits de traduction, d'adaptation et de reproduction réservés.

L'API NATIVE

Arnold McDonald (AMcD)

Version 1.01 – 1^{er} octobre 2002

L'API Native

« Il y a tant à apprendre qu'on ne pourrait dire de personne qu'il sait tout. »

Arthur Conan Doyle, *Sir Nigel*.

Résumé. *Cette fois-ci, je vous emmène à la découverte de ntdll.dll, la célèbre API « secrète » de Windows NT/2K/XP...*

Microsoft a très souvent été soupçonnée de dissimuler aux utilisateurs « certaines » caractéristiques de leurs produits... d'où l'inévitable émergence de rumeurs de toutes sortes¹ ! Par exemple, la firme fût, à une époque, l'objet d'une assez grave accusation : dans certains logiciels, des fonctionnalités « cachées » auraient un rôle plutôt « trouble » vis-à-vis de notre vie privée (mouchard ? lien avec la NSA² ?)...

Même les développeurs doivent faire avec ces dissimulations ! Certaines fonctions de Win32 (entre autre) n'ont jamais été documentées, ce qui a induit un sentiment d'inégalité par rapport à Microsoft, beaucoup se demandant si, par hasard, ces fonctions n'avantageaient pas « déloyalement » leurs applications maison³...

L'API secrète...

Il s'est même parfois dit que des APIs « entières » étaient cachées ! Depuis la sortie de Windows NT, le leader incontesté de cet oui-dire est la fameuse API « **native** », dont l'implémentation réside dans le fichier ntdll.dll.

ntdll.dll

Cette bibliothèque dynamique est distribuée avec Windows NT, 2000 et XP. Elle permet d'interfacer les services systèmes fournis par l'OS. Rien de bien magique ? Hum, voire...

Toutefois, cette DLL n'est que très rarement mentionnée dans les divers outils de développement de la firme de Redmond et son contenu n'a jamais été documenté. Ou très peu ! On trouve une poignée (24) de fonctions décrites dans le DDK⁴, de timides évocations sont faites dans la *Knowledge Base* (KB) ou le MSDN... Bref, pas

¹ Parfois parfaitement fondées d'ailleurs... puisque reconnues publiquement !

² *National Security Agency* (Agence de Sécurité Nationale). Organisme américain au budget pharaonique, dont le rôle exact est tellement nébuleux qu'il en fait la cible idéale de toute psychose à la « Big Brother »...

³ Qui a dit que les programmeurs étaient paranoïaques ?

⁴ *Driver Development Kit* (kit de développement pour *drivers*).

grand-chose à se mettre sous la dent ! Quant aux sources « externes », seuls quelques articles pour experts évoquent, parfois, une fonction de cette bibliothèque...

Aussi, inutile de préciser que la dénomination « API native » n'a rien d'officiel ! Il s'agit cependant d'une sorte de « consensus » dans le monde de la programmation Windows ; pour ma part, je ne l'ai jamais vue désignée autrement.

Parano

Évidemment, la raison d'être de cette DLL ainsi que le manque d'information à son sujet l'ont rendu « intéressante ». Surtout que, si nous ne sommes pas tenus au courant de son « intérêt », Windows en use, lui, sans vergogne ! Il suffit d'ailleurs de simplement regarder les tables d'imports de kernel32.dll ou user32.dll pour voir que des dizaines d'appels sont faits vers ntdll.dll...

Comment ça, Microsoft l'utilise à profusion et nous ne dit rien ? C'est louche ça... Le bruit a alors couru que certaines de ses fonctions permettaient des choses incroyables comme, par exemple, outrepasser certaines « sécurités »...

En réalité, il n'y a pas autant de mystère !

La raison du « complot »

Il faut tout d'abord se souvenir que Windows NT⁵ a été conçu pour pouvoir exécuter divers types d'applications : les « traditionnels » Win32, Win16 et DOS, certes, mais également OS/2 et POSIX⁶. Pour permettre cela, une structure bien particulière a été adoptée :

- L'architecture de NT repose sur deux modes fondamentaux :
 - Le **mode user** (utilisateur). Celui où s'exécutent les applications. Il utilise le niveau de privilège 3 du processeur et ne permet pas de s'adresser directement aux périphériques. L'accès aux données du système est limité et contrôlé.
 - Le **mode kernel** (noyau⁷). Celui dans lequel s'exécute le système d'exploitation. Il utilise le niveau de privilège 0 du processeur, où, pour faire simple, tout est permis !

Il s'agit là de la première étape : une séparation franche entre les applications et l'OS, ce qui assure, en théorie, une meilleure robustesse du système. Un réel progrès par rapport à Windows 95/98/Me !

⁵ À partir d'ici, je ferai référence à l'acronyme NT pour désigner Windows NT, 2000 ou XP.

⁶ *Portable Operating System Interface based on UNIX*.

⁷ Le terme *kernel* est rarement traduit dans la littérature étrangère. Par exemple, je n'ai jamais vu l'équivalent du mot « noyau » dans un texte russe ou allemand...

- Le « cœur » de NT est constitué de trois parties⁸ :
 - L'**exécutif**. Contient tous les services et fonctions « de base » du système d'exploitation. Par exemple, la gestion de la mémoire ou celle des *threads* et des processus.
 - Le **kernel**. Y résident les fonctions dites de « bas niveau », comme la gestion des exceptions et des interruptions.
 - Les **drivers**. Ce terme sert à désigner les *drivers* de périphériques, mais aussi ceux de réseau ou de système de fichier.

C'est la seconde étape : la « segmentation » des différents composants du système d'exploitation en différentes couches, ce qui, entre autre, permet de faciliter la portabilité vers différentes plates-formes matérielles.

- Les applications ne peuvent pas appeler directement les services de NT. Elles « passent » par un **sous-système d'environnement (SSE)**, dont le rôle principal est de mettre à leur disposition une API⁹ pour accéder, de manière indirecte, aux fonctions natives du système d'exploitation.

Windows NT est livré avec trois SSEs : Win32, celui dit « natif¹⁰ », OS/2 et POSIX. Un SSE actif est associé à un processus système. Par exemple, celui de Win32 est le fameux *csrss.exe*¹¹ qui apparaît dans la liste des processus du gestionnaire des tâches (appelé par CTRL+ALT+DEL).

Cette ultime étape permet de disposer, via les SSEs, de plusieurs « OS ».

C'est ici qu'est susceptible d'intervenir notre *ntdll.dll*. Lorsqu'une application a une requête à effectuer, elle appelle une fonction d'une des bibliothèques de l'API de son sous-système d'environnement. Trois cas se présentent alors :

- La fonction de la DLL impliquée peut satisfaire les besoins de l'appelant, sans aucune autre « ressource » extérieure, que ce soit de NT ou du SSE. La fonction s'exécute et retourne.
- La fonction nécessite de faire intervenir son sous-système d'environnement. Des échanges de messages vont alors avoir lieu entre la DLL (le client) et le processus du SSE (le serveur¹²).

⁸ Je simplifie au maximum ! Les « puristes » se référeront aux ouvrages cités en référence p. 19–20.

⁹ Dont l'implémentation effective se présente sous la forme de DLLs (qui s'exécutent en mode *user*).

¹⁰ Ses DLLs fondamentales sont *advapi32.dll*, *gdi32.dll*, *kernel32.dll* et *user32.dll*.

¹¹ *Client/Server Run-time SubSystem*.

¹² Cette forme de communication est l'origine de la qualification « client/serveur » de NT.

- La fonction a besoin de l'aide de l'exécutif de NT. L'interface vers les services systèmes étant ntdll.dll... on les appelle **via** cette bibliothèque ! En clair, les DLLs d'API d'un SSE servent à « traduire » les appels de leurs fonctions (documentées) en appels (via ntdll.dll, donc) de services système de l'exécutif¹³ (non documentés)...

À ce stade de la discussion, avouez qu'il semble utile de résumer tout cela sur une figure ! Pour faire « léger », on traite seulement le cas d'une application Win32 :

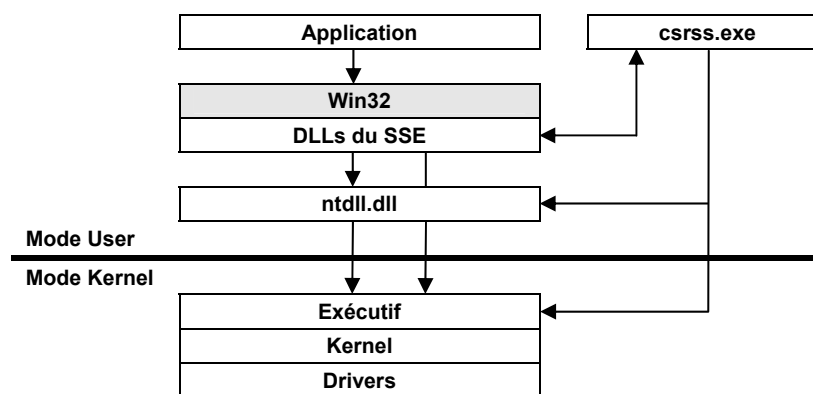


Figure 1. L'architecture, très simplifiée, de NT.

Il faut, en outre, préciser que, pour des raisons d'optimisation des performances, gdi32.dll et user32.dll passent principalement par win32k.sys (non représenté sur la Fig. 1), *driver* s'exécutant en mode *kernel*. Ce n'était pas le cas avant NT4, où on utilisait le serveur csrss.exe ; le moins qu'on puisse dire, c'est que les opérations graphiques s'en ressentaient...

Résumons !

La « secrète » ntdll.dll n'est donc rien d'autre que l'implémentation d'une API qui permet aux applications en mode *user* d'accéder aux services systèmes de l'exécutif de NT (d'où sa qualification de « native »). Il n'y a donc rien de si... mystérieux !

De plus, si les fonctions de cette DLL ne sont pas documentées, ce n'est pas si grave en soi, puisque, si on y réfléchit bien, on y « accède » au moyen des fonctions des APIs des sous-systèmes d'environnement, qui le sont, elles, abondamment ! En clair, il suffit de se renseigner sur la fonction d'API pour « soupçonner » le service éventuellement appelé par ntdll.dll¹⁴...

¹³ D'où la raison de tant d'appels vers ntdll.dll depuis les DLLs de Win32...

¹⁴ De toute façon, il faut maîtriser le développement via les APIs « classiques » avant de s'attaquer à ntdll.dll !

Alors ? Seuls quelques « passionnés » de technique seraient en droit de déplorer que Microsoft n'ait pas fait l'effort de documenter cette API « native » de services système¹⁵ ? Malheureusement non...

Il y a, tout de même, un petit problème !

Car, en réalité, toutes les fonctions de ntdll.dll ne sont pas « encapsulées » par les fonctions d'API d'un SSE ; certaines sont destinées à être uniquement utilisées en « interne » par Windows¹⁶ et permettent des choses **impossibles** à réaliser via l'utilisation des APIs classiques, comme, par exemple, obtenir la liste des *handles* ouverts d'un processus, récupérer le nombre de changements de contexte effectués par un processeur, ou fixer les paramètres étendus d'un fichier.

C'est cet aspect « réservé » qui a engendré la rumeur « d'avantage déloyal »...

Il n'y a d'ailleurs pas que le problème des fonctions « réservées » ! Il y a aussi celui de celles qui, bien que « documentées » via une fonction d'API, offrent de subtiles nuances d'utilisation bien pratiques si on utilise directement la version native (non documentée donc, à de très rares exceptions près).

Prenons le cas de CreateFile(), de Win32, qui se contente de 7 paramètres d'appel ; son « alter ego » de ntdll.dll, NtCreateFile(), en nécessitant 11... vous vous doutez bien que cela doit autoriser un plus grand nombre de possibilités¹⁷ !

Les domaines de Windows concernés par les fonctions « cachées/réservées » sont nombreux, même s'ils restent, tout de même, plutôt « spécifiques » :

- Objets système.
- LPC (*Local Procedure Calls*).
- Mémoire virtuelle.
- Informations sur le système.
- Sécurité.
- Etc.

Il y a plusieurs dizaines de fonctions à découvrir (redécouvrir ?) et un livre entier n'y suffirait pas ! Cet article étant juste une introduction sur le sujet, si l'API native vous passionne je vous conseille alors de bondir sur les ouvrages cités en référence, notamment [2000NEBB].

¹⁵ Comme cela l'est pour d'autres OS, UNIX, par exemple...

¹⁶ Notamment [2000SOLO], p. 60.

¹⁷ Plus d'options de création, de spécifications d'attributs de fichiers, etc.

Avantage déloyal ?

Oui !

Dans le sens où certaines fonctions sont « irremplaçables »... puisqu'il est presque impossible d'accomplir la même chose sans passer par elles ! Voir l'exemple à la fin de cet article.

Souvent, pour manipuler des objets en mode *kernel*, la seule solution est de passer par l'écriture d'un *driver* ; c'est quand même bien plus pratique et facile d'utiliser une fonction déjà toute prête !

Ensuite, appeler directement l'API native peut, dans quelques cas, s'avérer plus rapide, puisqu'on court-circuite la surcharge de traitement inhérente au « contrôle » de notre application par un sous-système d'environnement. Il est donc possible, sous certaines conditions, d'obtenir un code plus véloce.

Pourquoi alors *ntdll.dll* n'est-elle pas « publique » ? Il est incontestable que, vu le « sujet » concerné par ces fonctions cachées/réservées, il serait, par exemple, bien plus facile d'écrire des utilitaires système efficaces, à peine moins puissants que ceux fournis par Microsoft¹⁸...

Non !

Pour au moins cinq raisons :

- Microsoft encourage l'utilisation d'une API de SSE, comme Win32, pour écrire des applications... et n'offre aucun support pour les fonctions de *ntdll.dll*. Si vous ne souhaitez pas produire du code peut-être demain obsolète ou simplement non exécutable, utilisez les API classiques !
- Les besoins d'une application « standard », vis-à-vis de NT, peuvent tous être satisfaits via les fonctions des APIs fournies par les sous-systèmes d'environnement.
- De par sa très faible documentation, l'emploi de l'API native est une tâche qui peut s'avérer bien vite longue et fastidieuse¹⁹. Si vous n'en avez pas absolument besoin, évitez-la !
- Les fonctions qui ne sont pas documentées concernent quand même un public « averti ». Soyons francs, si vous ne développez pas des utilitaires

¹⁸ Qui, je vous rassure, ne se prive pas d'utiliser ces fonctions « secrètes » !

¹⁹ Autant préciser tout de suite aux « courageux » qu'il leur sera souvent indispensable de s'adonner aux joies de l'ingénierie inverse pour comprendre un paramètre d'appel, un retour de fonction, etc.

système ou des outils de débogage, vous accorderez certainement un intérêt plus que limité à l'API native...

- Dites-vous bien que le contenu de ntdll.dll n'est pas fixé de manière définitive ! À chaque évolution du noyau de NT, vous pouvez être sûr que des modifications vont apparaître au niveau de cette bibliothèque.

À titre d'exemple, le tableau suivant recense quelques variations subies par cette DLL avec les diverses versions de Windows :

OS	Version de ntdll	Octets	Fonctions	
			Exportées	Services ²⁰
Windows 95	4.0.950	5.632	52	2
Windows NT4	4.0.1376.1	361.232	1.044	211
Windows 98	4.10.1998	20.480	52	8
Windows Me	4.90.3000	20.480	52	8
Windows 2K	5.0.2121.1	501.520	1.199	258
Windows XP	5.1.2600.0	699.392	1.408	285

Tableau 1. *Évolution de ntdll au cours des âges...*

Remarque Même si leur architecture n'a strictement rien à voir avec celle de NT, Windows 95, 98 et Me sont aussi livrés avec un fichier ntdll.dll. Il contient surtout des routines « d'exécution²¹ ».

Il n'y a pas que le nombre de fonctions qui change ! Les développeurs consulteront le fichier ntddk.h des versions successives du DDK pour constater que les structures de données évoluent également. Bref, autant dire qu'utiliser l'API native dans une application implique l'impossibilité d'en garantir simultanément la pérennité...

Conclusion

La même que pour toute rumeur : il y a une part de vrai et une part de faux ! Le « faux », c'est que ntdll.dll n'est pas une bibliothèque au rôle si secret ou magique que cela. De toute façon, elle ne présente que peu d'intérêt pour un développeur « classique », qui utilise d'abord les fonctions « recommandées » de son SSE favori.

²⁰ Fonctions préfixées par les lettres Nt (*Native*).

²¹ Dont le nom est préfixé par Rtl (*Run-time Library*).

Le « vrai », c'est qu'une partie non négligeable des fonctions de `ntdll.dll` ne possède pas d'équivalent dans les APIs fournies pour programmer les SSE²² et n'est pas, de plus, documentée. Un développeur d'utilitaires système peut donc se considérer réellement désavantagé vis-à-vis de Microsoft et des sociétés que la toute puissante entreprise a favorisées en les informant²³...

Terminons plutôt sur une question que vous devriez maintenant logiquement vous poser : y a-t-il d'autres APIs « cachées » ? Heu...

Exemple

Avant de nous quitter, je vais vous montrer la « puissance » de cette fameuse API native au travers d'un petit exemple, appelé... « Native » : nous allons implémenter un « classique » du genre, une application qui affiche la liste de ses *handles* ouverts.

Inspection de `ntdll.dll`

Pour commencer, jetons un œil sur la DLL qui est fournie avec mon Windows XP « Édition familiale ». Elle contient 1.408 exports de fonctions, réparties ainsi :

Préfixe	Signification	Fonctions	Préfixe	Signification	Fonctions
Csr	Client/Server	15	Pfx	Prefix	4
Dbg	Debug	18	Rtl, Rtlp, Rtlx	Run-time library	511
Ki	Kernel Interrupt	4	Zw		284
Ldr	Loader	33	—		52
Nls	National Langage Support	3	—		115
Nt	Native	285	Sans préfixe		84

Tableau 2. Détail des fonctions exportées par `ntdll.dll` de Windows XP.

Il y a deux types de services dans `ntdll.dll` : les fonctions préfixées par `Nt` et celles préfixées par `Zw`²⁴. Ce sont les mêmes ! La différence entre les deux est, pour faire simple, que les routines « `Nt` » vérifient les paramètres d'appel et tiennent compte des privilèges. La raison en est que les services système peuvent être appelés depuis du code en mode *user*, mais également depuis du code en mode *kernel*, pour lequel, évidemment, le « contrôle » doit être moindre²⁵. D'où, deux jeux « parallèles » de points d'entrées...

²² Environ 40 % des fonctions natives du `ntdll.dll` de mon Windows XP. Tout de même !

²³ Autre rumeur visant une société développant un défragmenteur de disque...

²⁴ J'avoue que je sèche lamentablement quant à la signification de ces deux lettres !

²⁵ Il semble légitime de faire confiance à des développeurs ayant été « autorisés » à écrire du code *kernel*...

On trouve plusieurs sortes de fonctions parmi les services, entre autre :

- Celles pour lesquelles Win32 fournit une fonction d'appel « officielle ». Par exemple, ntdll.dll exporte NtGetContextThread(), de prototype²⁶ :

```
NTSYSAPI
NTSTATUS
NTAPI
NtGetContextThread(
    IN HANDLE ThreadHandle,
    OUT PCONTEXT Context
);
```

Dans les fichiers d'en-tête du SDK ou du DDK, on trouve :

```
#if !defined( NTSYSTEM )
#define NTSYSAPI DECLSPEC_IMPORT
#else
#define NTSYSAPI
#define DECLSPEC_IMPORT __declspec(dllimport)
```

NTSYSAPI sert donc à importer une fonction d'une DLL.

```
typedef LONG NTSTATUS
```

NTSTATUS précise donc le type retourné par la fonction.

```
#define NTAPI __stdcall
```

NTAPI permet donc de spécifier la convention d'appel de la fonction.

```
#define IN
#define OUT
```

Ces deux définitions sont très utilisées dans le DDK et servent à spécifier si un paramètre est utilisé en entrée ou en sortie.

On peut donc réécrire le prototype de NtGetContextThread() :

```
__declspec(dllimport)
LONG
__stdcall
NtGetContextThread(
    HANDLE ThreadHandle,
    PCONTEXT Context
);
```

²⁶ J'utilise le style et les conventions de [2000NEBB].

Comme on trouve également :

```
typedef int BOOL;  
typedef long LONG;
```

En « supposant » qu'un `int` et un `long` ont la même taille, on obtient donc le même prototype que celui de `GetThreadContext()`, proposé par l'API Win32 :

```
BOOL  
GetThreadContext (  
    HANDLE        hThread,  
    LPCONTEXT27 lpContext  
);
```

Tout juste remarque-t-on l'inversion de mots dans le nom de la fonction, chose d'ailleurs très fréquente lorsqu'on compare les prototypes des fonctions de `ntdll.dll` et ceux de l'API Win32.

- Les fameuses fonctions cachées/réservées non documentées ! Bien que très largement utilisées par Windows... ou les routines de Win32, aucun équivalent « direct » ne nous est proposé dans les APIs classiques.

Par exemple, `NtQueryInformationProcess()` ne possède pas de fonction comparable dans l'API Win32. Or, il suffit de désassembler `advapi32.dll`, ou `user32.dll`, pour repérer de multiples références vers cette routine... et se « douter » de sa grande utilité !

- Des services visiblement peu utilisés. Par exemple, je n'ai pas trouvé de DLL de Win32 appelant `NtShutdownSystem()` ou `NtPulseEvent()`.

Utilisation de l'API native

Généralement, l'interfaçage d'une DLL nécessite un fichier d'en-tête, d'extension `.h`, qui contient les prototypes des fonctions de la bibliothèque et un fichier `.lib`, qui contient la liste des symboles exportés, indispensable pour pouvoir lier l'exécutable à la DLL.

Le hic, c'est que comme l'API native est non documentée, Microsoft ne fournit pas ces fichiers `.h/.lib` pour `ntdll.dll`²⁸... Il faut donc faire sans et passer par les fonctions `LoadLibrary()` et `GetProcAddress()`.

²⁷ Ne vous inquiétez pas, on trouve bien `typedef PCONTEXT LPCONTEXT;` dans le SDK...

²⁸ Les rares fonctions natives « documentées » peuvent toutefois s'interfacer via `ntddk.h` et `ntddk.lib` du DDK.

Bien sûr, il va nous falloir redéfinir les structures, les déclarations de prototypes, de fonctions, etc. Je vous conseille ardemment de vous procurer l'indispensable ouvrage de Gary Nebbett cité en référence !

Le programme

Sa structure est simplissime ! On utilise une boîte de dialogue qui affiche, via un contrôle « ListView », la liste de tous les *handles* du processus de notre application.

Fonctions natives

Deux sont requises :

- `NtQuerySystemInformation()`. La fonction élémentaire pour récupérer des informations sur le système. Voici son prototype « simplifié » :

```
NTSYSAPI
NTSTATUS
NTAPI
NtQuerySystemInformation (
    SYSTEM_INFORMATION_CLASS siClass,
    PVOID                    siBuff,
    ULONG                   siBuffLength,
    PULONG                  ReturnLength
);
```

`siClass` : Type d'information que l'on souhaite récupérer. En fait, il s'agit d'un index dans une liste d'énumération de type `ULONG`.

`siBuff` : Tampon recevant les informations.

`siBuffLength` : Taille du tampon.

`ReturnLength` : Reçoit le nombre d'octets effectivement retournés.

- `NtQueryObject()`. Utilisée pour obtenir des informations sur un objet du système. Ses paramètres ont le même « rôle » que ceux de la fonction `NtQuerySystemInformation()`. La seule différence est qu'il faut spécifier le *handle* de l'objet qui nous intrigue :

```
NTSYSAPI
NTSTATUS
NTAPI
NtQueryObject (
    HANDLE                   hObject,
    OBJECT_INFORMATION_CLASS oiClass,
    PVOID                   oiBuff,
    ULONG                   oiBuffLength,
    PULONG                  ReturnLength
);
```

Code source

Afin de contenir cet article dans des longueurs raisonnables, quelques conventions ont été adoptées pour l'écriture du programme :

- Gestion des erreurs inexistante ! Il s'agit d'un code servant à illustrer un article et il n'a pas pour vocation de servir en milieu professionnel...
- Aucun commentaire, les opérations effectuées étant quand même plutôt triviales ! Toutefois, certains points qui méritent d'être un peu détaillés le sont en p. 15.

```
001 #define UNICODE
002
003 #include <windows.h>
004 #include <commctrl.h>
005 #include <ntsecapi.h>
006 #include "resource.h"
007
008 #define OBJECT_NAME_INFORMATION 1
009 #define OBJECT_TYPE_INFORMATION 2
010 #define SYSTEM_HANDLE_INFORMATION 16
011 #define STATUS_INFO_LENGTH_MISMATCH 0xC0000004L
012
013 typedef ULONG (__stdcall* NTQUERYSYSTEMINFORMATION) (ULONG, PVOID, ULONG, PULONG);
014 typedef ULONG (__stdcall* NTQUERYOBJECT) (HANDLE, ULONG, PVOID, ULONG, PULONG);
015
016 typedef struct _SYSTEMHANDLEINFORMATION
017 {
018     ULONG ProcessId;
019     UCHAR ObjectTypeNumber;
020     UCHAR Flags;
021     USHORT Handle;
022     PVOID Object;
023     ACCESS_MASK GrantedAccess;
024 }
025 SYSTEMHANDLEINFORMATION, *PSYSTEMHANDLEINFORMATION;
026
027 typedef struct _OBJECTNAMEINFORMATION
028 {
029     UNICODE_STRING Name;
030 }
031 OBJECTNAMEINFORMATION, *POBJECTNAMEINFORMATION;
032
033 typedef struct _OBJECTTYPEINFORMATION
034 {
035     UNICODE_STRING Name;
036     ULONG ObjectCount;
037     ULONG HandleCount;
038     ULONG Reserved1[4];
039     ULONG PeakObjectCount;
040     ULONG PeakHandleCount;
041     ULONG Reserved2[4];
042     ULONG InvalidAttributes;
043     GENERIC_MAPPING GenericMapping;
044     ULONG ValidAccess;
045     UCHAR Unknown;
046     BOOLEAN MaintainHandleDataBase;
047     ULONG PoolType;
048     ULONG PagedPoolUsage;
```



```

049  ULONG          NonPagedPoolUsage;
050  }
051  OBJECTTYPEINFORMATION,*POBJECTTYPEINFORMATION;
052
053  HINSTANCE  hAppInst;
054  HWND      hAppListView;
055
056  void Go(HWND hWdw)
057  {
058      BYTE          byBuff1[1024],byBuff2[1024];
059      DWORD         dwProcessID;
060      HINSTANCE     hntdll;
061      int           nIndex;
062      LVITEM        lvi;
063      NTQUERYOBJECT NtQueryObject;
064      NTQUERYSYSTEMINFORMATION NtQuerySystemInformation;
065      PBYTE         pbyBuff;
066      POBJECTNAMEINFORMATION poniObjectNameInfo;
067      POBJECTTYPEINFORMATION potiObjectTypeInfo;
068      PSYSTEMHANDLEINFORMATION psiHandleInfo;
069      PULONG        plBuff;
070      SYSTEMHANDLEINFORMATION siHandleInfo;
071      TCHAR         sz[5][256];
072      ULONG         lBuffEltNb;
073      ULONG         lRet;
074      ULONG         lSystemHandleNb;
075      ULONG         lProcessHandleNb;
076      ULONG         i,j;
077
078      hntdll = LoadLibrary(TEXT("ntdll.dll"));
079
080      NtQuerySystemInformation = (NTQUERYSYSTEMINFORMATION)GetProcAddress(hntdll,
                                                                              "NtQuerySystemInformation");
081      NtQueryObject           = (NTQUERYOBJECT)GetProcAddress(hntdll,
                                                                              "NtQueryObject");
082
083      lBuffEltNb = 1;
084
085      plBuff = (PULONG)GlobalAlloc(GMEM_FIXED,lBuffEltNb*sizeof(ULONG));
086
087      while ( STATUS_INFO_LENGTH_MISMATCH ==
              NtQuerySystemInformation(SYSTEM_HANDLE_INFORMATION,
              plBuff,
              lBuffEltNb*sizeof(ULONG),
              &lRet) )
088      {
089          lBuffEltNb += 256;
090
091          GlobalFree(plBuff);
092
093          plBuff = (PULONG)GlobalAlloc(GMEM_FIXED,lBuffEltNb*sizeof(ULONG));
094      }
095
096      lSystemHandleNb = (ULONG)*plBuff;
097
098      plBuff += 1;
099
100      psiHandleInfo = (PSYSTEMHANDLEINFORMATION)plBuff;
101
102      GetWindowThreadProcessId(hWdw,&dwProcessID);
103
104      lProcessHandleNb = 0;
105
106      for ( i = 0; i < lSystemHandleNb; i++ )
107      {

```

```

108     siHandleInfo = *psiHandleInfo;
109
110     if ( dwProcessID == siHandleInfo.ProcessId )
111     {
112         pbyBuff = byBuff1;
113
114         NtQueryObject((HANDLE)siHandleInfo.Handle,
115                     OBJECT_NAME_INFORMATION,
116                     pbyBuff,
117                     sizeof(byBuff1),
118                     &lRet);
119
120         poniObjectNameInfo = (POBJECTNAMEINFORMATION)pbyBuff;
121
122         pbyBuff = byBuff2;
123
124         NtQueryObject((HANDLE)siHandleInfo.Handle,
125                     OBJECT_TYPE_INFORMATION,
126                     pbyBuff,
127                     sizeof(byBuff2),
128                     &lRet);
129
130         potiObjectTypeInfo = (POBJECTTYPEINFORMATION)pbyBuff;
131
132         wsprintf(sz[0],TEXT("%04d"),++lProcessHandleNb);
133
134         wsprintf(sz[1],TEXT("0x%04X"),siHandleInfo.Handle);
135
136         wsprintf(sz[2],TEXT("%s"),potiObjectTypeInfo->Name.Buffer);
137
138         wsprintf(sz[3],TEXT("0x%08X"),siHandleInfo.GrantedAccess);
139
140         wsprintf(sz[4],TEXT("%s"),poniObjectNameInfo->Name.Buffer);
141
142         memset(&lvi,0,sizeof(lvi));
143
144         lvi.mask     = LVIF_TEXT;
145         lvi.iItem    = lProcessHandleNb;
146         lvi.pszText = sz[0];
147
148         nIndex = ListView_InsertItem(hAppListView,&lvi);
149
150         for ( j = 1; j < 5; j++ )
151         {
152             lvi.iItem    = nIndex;
153             lvi.iSubItem = j;
154             lvi.pszText = sz[j];
155
156             ListView_SetItem(hAppListView,&lvi);
157         }
158     }
159     psiHandleInfo += 1;
160 }
161
162 BOOL CALLBACK DlgProc(HWND hWdw,UINT uiMess,WPARAM wParam,LPARAM lParam)
163 {
164     DWORD    dwStyle = WS_CHILD|WS_VISIBLE|LVS_REPORT|LVS_SINGLESEL|WS_BORDER;

```

```
165 int i;
166 int nColCX[] = { 35,50,90,75,400 };
167 LVCOLUMN lvc;
168 PTSTR szColTitle[] = { TEXT("No"),TEXT("Handle"),TEXT("Type"),
                        TEXT("Accès"),TEXT("Nom") };
169
170 switch( uiMess )
171 {
172     case WM_INITDIALOG : hAppListView = CreateWindow(WC_LISTVIEW,
                                                    TEXT(""),
                                                    dwStyle,
                                                    0,
                                                    0,
                                                    651,
                                                    306,
                                                    hWdw,
                                                    (HMENU)1000,
                                                    hAppInst,
                                                    NULL);
173
174     memset(&lvc,0,sizeof(lvc));
175
176     lvc.mask = LVCF_FMT|LVCF_TEXT|LVCF_SUBITEM|LVCF_WIDTH;
177     lvc.fmt = LVCFMT_LEFT;
178
179     for ( i = 0; i < 5; i++ )
180     {
181         lvc.cx = nColCX[i];
182         lvc.pszText = szColTitle[i];
183         lvc.iSubItem = i;
184
185         ListView_InsertColumn(hAppListView,i,&lvc);
186     }
187
188     dwStyle = (DWORD)SendMessage(hAppListView,
                                LVM_GETEXTENDEDLISTVIEWSTYLE,
                                0,
                                0);
189
190     dwStyle |= LVS_EX_FULLROWSELECT|LVS_EX_GRIDLINES;
191
192     SendMessage(hAppListView,
                LVM_SETEXTENDEDLISTVIEWSTYLE,
                0,
                dwStyle);
193
194     Go(hWdw);
195
196     return TRUE;
197
198     case WM_CLOSE : EndDialog(hWdw,0);
199
200     return TRUE;
201
202     default : return FALSE;
203 }
204 }
205
206 int WINAPI WinMain(HINSTANCE hInst,
                    HINSTANCE hPrevInst,
                    LPSTR pszCmdLine,
                    int nCmdShow)
207 {
208     InitCommonControls();
209 }
```

```
210  hAppInst = hInst;
211
212  DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL, DlgProc);
213
214  return 0;
215 }
```

Figure 2. Code source du programme Native.

Quelques commentaires...

Ligne 5 : Le fichier ntsecapi.h contient la définition du type UNICODE_STRING, que l'on trouve dans les structures OBJECTXXXINFORMATION.

Lignes 8-11 : Définitions de constantes utilisées par les deux fonctions natives.

Lignes 13-14 : « Types » des prototypes des fonctions natives NtQueryObject() et NtQuerySystemInformation().

Lignes 16-51 : Structures utilisées par les fonctions natives pour stocker les informations sur les objets système.

Lignes 53-54 : Variables globales.

Lignes 83-94 : NtQuerySystemInformation() ne retourne « correctement » que si le *buffer* passé en paramètre est de taille suffisante pour contenir les informations recherchées. On « boucle » donc jusqu'à ce que le tampon `p1Buff` soit assez grand pour recevoir la liste de tous les *handles* du système. À chaque nouvelle allocation, on incrémente la taille du *buffer* de 256 ULONG²⁹.

Ligne 96 : Arrivé ici, le premier ULONG de `p1Buff` contient le nombre exact de *handles* effectivement chargés. Vient ensuite la série de structures qui contiennent les informations sur les *handles*.

Lignes 98-100 : On passe à la première structure SYSTEMHANDLEINFORMATION.

Ligne 102 : Récupération de l'identificateur du processus de notre application. En effet, `p1Buff` contient la liste de **tous** les *handles* du système, or, seuls ceux appartenant à notre application nous intéressent. On les différenciera des autres à l'aide du champ `ProcessId` de la structure SYSTEMHANDLEINFORMATION.

Lignes 106-153 : On parcourt la liste des *handles* système ; si on en trouve un qui appartient à notre processus, on ajoute une ligne le concernant dans le contrôle de type LISTVIEW de la boîte de dialogue de l'application. Les informations sur le *handle* sont obtenues via deux appels à NtQueryObject().

Le reste du code ne devrait pas poser de problème...

²⁹ J'ai choisi la valeur 256 de façon absolument arbitraire !

Autres fichiers

Il ne reste plus qu'à présenter le fichier d'identificateurs de ressources :

```

01 //{NO_DEPENDENCIES}
02 // Microsoft Visual C++ generated include file.
03 // Used by Native.rc
04 //
05 #define IDD_DIALOG1                101
06
07 // Next default values for new objects
08 //
09 #ifndef APSTUDIO_INVOKED
10 #ifndef APSTUDIO_READONLY_SYMBOLS
11 #define _APS_NEXT_RESOURCE_VALUE    102
12 #define _APS_NEXT_COMMAND_VALUE    40001
13 #define _APS_NEXT_CONTROL_VALUE    1001
14 #define _APS_NEXT_SYMED_VALUE      101
15 #endif
16 #endif
17

```

Figure 3. Le contenu de *resource.h*.

Et voici le fichier de ressources lui-même !

```

// Microsoft Visual C++ generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//
// Français (France) resources
//
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_FRA)
#ifdef _WIN32
LANGUAGE LANG_FRENCH, SUBLANG_FRENCH
#pragma code_page(1252)
#endif // _WIN32

#ifdef APSTUDIO_INVOKED
//
// TEXTINCLUDE
//
1 TEXTINCLUDE
BEGIN
    "resource.h\0"

```

```
END

2 TEXTINCLUDE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

/////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_DIALOG1 DIALOGEX 0, 0, 434, 188
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_CAPTION |
    WS_SYSMENU
CAPTION "Native - (c) 2002 by AMcD/Quack"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
BEGIN
END

/////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 327
        TOPMARGIN, 7
        BOTTOMMARGIN, 181
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // Français (France) resources

/////////////////////////////////////////////////////////////////

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
```

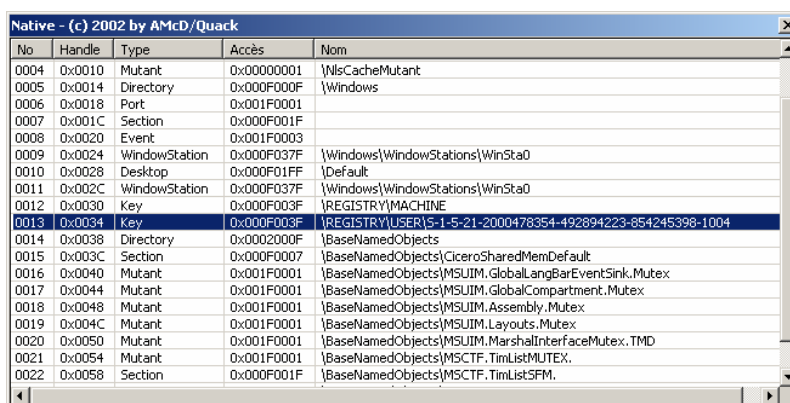
```
//  
  
////////////////////////////////////  
  
#endif // not APSTUDIO_INVOKED
```

Figure 4. Le fichier *Native.rc*.

C'est d'ailleurs l'occasion de constater, une fois encore, la lourdeur des fichiers de ressources Windows ! Tout ce bric-à-brac est-il vraiment nécessaire pour déclarer une bête boîte de dialogue ?

Une dernière remarque ! Avant de m'inonder de mails réprobateurs, vérifiez bien que vous liez le tout avec le fichier *comctl32.lib*, utilisé par le contrôle liste de la boîte de dialogue...

Enfin, pour ceux qui ne se sentent pas capables de fournir l'effort de compiler tout ce bazar, voici à quoi ressemble le résultat final en cours d'exécution :



No	Handle	Type	Accès	Nom
0004	0x0010	Mutant	0x00000001	\NlsCacheMutant
0005	0x0014	Directory	0x000F000F	\Windows
0006	0x0018	Port	0x001F0001	
0007	0x001C	Section	0x000F001F	
0008	0x0020	Event	0x001F0003	
0009	0x0024	WindowStation	0x000F037F	\Windows\WindowStations\WinSta0
0010	0x0028	Desktop	0x000F01FF	\Default
0011	0x002C	WindowStation	0x000F037F	\Windows\WindowStations\WinSta0
0012	0x0030	Key	0x000F003F	\REGISTRY\MACHINE
0013	0x0034	Key	0x000F003F	\REGISTRY\USER\5-1-5-21-2000478354-492894223-854245398-1004
0014	0x0038	Directory	0x0002000F	\BaseNamedObjects
0015	0x003C	Section	0x000F0007	\BaseNamedObjects\CiceroSharedMemDefault
0016	0x0040	Mutant	0x001F0001	\BaseNamedObjects\MSUIM.GlobalLangBarEventSink.Mutex
0017	0x0044	Mutant	0x001F0001	\BaseNamedObjects\MSUIM.GlobalCompartment.Mutex
0018	0x0048	Mutant	0x001F0001	\BaseNamedObjects\MSUIM.Assembly.Mutex
0019	0x004C	Mutant	0x001F0001	\BaseNamedObjects\MSUIM.Layouts.Mutex
0020	0x0050	Mutant	0x001F0001	\BaseNamedObjects\MSUIM.MarshalInterfaceMutex.TMD
0021	0x0054	Mutant	0x001F0001	\BaseNamedObjects\MSCTF.TimListMUTEX.
0022	0x0058	Section	0x000F001F	\BaseNamedObjects\MSCTF.TimList5FM.

Figure 5. La mirifique application *Native.exe...*

Suggestions

Si ce programme vous inspire, n'hésitez pas à y ajouter quelques fonctionnalités. Par exemple :

- Possibilité de fermer un *handle*.
- Affichage d'informations supplémentaires, par exemple, les attributs de sécurité du *handle*.

- Pouvoir modifier certaines propriétés de l'objet référencé par un *handle*³⁰.
- Montrer la liste de tous les processus actifs et afficher les *handles* de celui qui est sélectionné³¹.

Références

- [1999DABA] **Undocumented Windows NT**
Prasak Dabak, Sandeep Phadke & Milind Borate
M&T Books – 335 p.
- [2000NEBB] **Windows NT/2000 Native API Reference**
Gary Nebbett
New Riders Publishing – 495 p.
- [2000SOLO] **Inside Microsoft Windows 2000**
David Solomon & Mark Russinovich
Microsoft Press – 948 p.
- [2001SCHR] **Undocumented Windows 2000 Secrets**
Sven Schreiber
Addison Wesley – 592 p.

C'est terminé ! J'espère que tout cela vous sera utile. Comme d'habitude, n'hésitez pas à me faire parvenir vos critiques et remarques.

*Arnold McDonald (AMcD)*³²
amcd@club-internet.fr

³⁰ Oui, c'est un peu « chaud » à faire...

³¹ Voir l'application « Process Explorer » de Sysinternals.

³² Alias « le roi de la note de bas de page ! ».

Addendum

Suite à une intéressante conversation avec Gaëtan Semet, sur l'obligatoire groupe de nouvelles `fr.comp.os.ms-windows.programmation`, il m'apparaît utile d'apporter quelques commentaires supplémentaires sur le programme Native ; plus précisément, sur la partie située entre les lignes 83 et 94, rappelées ci-dessous :

```
083  lBuffEltNb = 1;
084
085  plBuff = (PULONG)GlobalAlloc(GMEM_FIXED, lBuffEltNb*sizeof(ULONG));
086
087  while ( STATUS_INFO_LENGTH_MISMATCH ==
          NtQuerySystemInformation(SYSTEM_HANDLE_INFORMATION,
                                   plBuff,
                                   lBuffEltNb*sizeof(ULONG),
                                   &lRet) )
088  {
089      lBuffEltNb += 256;
090
091      GlobalFree(plBuff);
092
093      plBuff = (PULONG)GlobalAlloc(GMEM_FIXED, lBuffEltNb*sizeof(ULONG));
094  }
```

Simplifions...

Il est en effet possible d'écrire plus simplement ce bout de code ! En fait, j'ai omis de vous préciser (p. 11) que si la taille du *buffer* passé en paramètre à la fonction native `NtQuerySystemInformation()` est insuffisante, `lRet` est censée retourner, normalement, le nombre d'octets nécessaires pour stocker l'information demandée. En clair, on peut donc supprimer la boucle `while`...

Oui mais...

Cependant, si vous essayez de déboguer Native, vous constaterez que le premier appel à `NtQuerySystemInformation()` retourne la valeur 0 dans `lRet`... et non la taille requise du tampon !

Dans la boucle `while`, ce n'est que lorsque `lBuffEltNb` atteint 257 que la bonne valeur nous est retournée dans `lRet`. Quel est ce mystère ? Le comportement de la variable `lRet` serait-il chaotique ?

Analysons !

En fait, si on incrémentait `lBuffEltNb` de 1 en 1, on constaterait que dès que la valeur 5 est atteinte, `lRet` retourne la bonne taille. 5 ? Cela correspond à une taille de *buffer* de $5 \times 4 = 20$ octets³³...

³³ Un ULONG occupe 4 octets.

La taille d'une structure SYSTEMHANDLEINFORMATION, le type des objets dont on cherche à obtenir la liste, est de 16 octets³⁴. Rappelez-vous (p. 16) qu'un ULONG figure également en tête de la liste d'objets lorsque NtQuerySystemInformation() s'est correctement exécutée. Or, 16+4 = ?

Conclusion ?

Pour que lRet contienne la taille exacte du tampon nécessaire au stockage des informations, il faut transmettre à NtQuerySystemInformation() un *buffer* dont la taille est **au moins** égale à celle d'une structure SYSTEMHANDLEINFORMATION... plus celle d'un ULONG !

Voici le code « simplifié » :

```
083 plBuff = (PULONG)GlobalAlloc(GMEM_FIXED,
                                sizeof(ULONG)+sizeof(SYSTEMHANDLEINFORMATION));
084
085 if ( STATUS_INFO_LENGTH_MISMATCH ==
        NtQuerySystemInformation(SYSTEM_HANDLE_INFORMATION,
                                plBuff,
                                sizeof(ULONG)+sizeof(SYSTEMHANDLEINFORMATION),
                                &lRet) )
086 {
087     GlobalFree(plBuff);
088
089     plBuff = (PULONG)GlobalAlloc(GMEM_FIXED,lRet);
090
091     NtQuerySystemInformation(SYSTEM_HANDLE_INFORMATION,plBuff,lRet,&lRet);
092 }
```

Avouez, quand même, que les codeurs de Microsoft sont parfois « bizarres »...

³⁴ Un ULONG (4 octets), 2 UCHAR (2 x 1), 1 USHORT (2), 1 PVOID (4) et 1 ACCESS_MASK (4).