Tworzenie i testowanie sterowników systemu Microsoft Windows™ 2000

Tomasz KREMPEL

Wydział Cybernetyki
Wojskowa Akademia Techniczna, ul. Kaliskiego 2, 00-908 Warszawa

STRESZCZENIE: W artykule omówiono sposób postępowania przy projektowaniu, implementacji i testowaniu sterowników urządzeń dla systemów operacyjnych rodziny Microsoft Windows. Ograniczono się do omówienia sterowników zgodnych z modelem WDM, przedstawiając mechanizmy obsługi urządzeń zewnętrznych. Ponadto przedstawiono w skrócie ewolucję sterowników w systemach operacyjnych firmy Microsoft.

1. Wprowadzenie

Obsługa urządzeń wejścia-wyjścia w systemach operacyjnych Microsoft Windows odbywa się za pomocą sterowników urządzeń. Sterownik urządzenia to segment oprogramowania będący zbiorem procedur, które wywoływane przez system wykonują dla niego określone operacje związane z urządzeniem. Sterowniki ładowane są do pamięci operacyjnej w trakcie inicjowania systemu i mogą pracować w jednym z dwóch trybów pracy systemu: jądra lub użytkownika. Urządzenia zewnętrzne obsługiwane są przede wszystkim przez sterowniki trybu jądra, mające bezpośredni dostęp do wszystkich zasobów systemu, co wymusza stosowanie odpowiedniej technologii ich tworzenia. Faza projektowania i implementacji sterowników jest procesem złożonym, po którym musi nastąpić ich weryfikacja i w razie konieczności debugowanie w celu wyśledzenia błędów.

Wczesne procesory komputerów klasy IBM PC udostępniały tylko jeden tryb pracy (tryb rzeczywisty), w którym pracowały programy użytkowników, jak również sterowniki urządzeń. Sterowniki występowały w postaci plików binarnych z rozszerzeniem .SYS i były ładowane do pamięci dzięki plikowi

wsadowemu CONFIG.SYS. Były to moduły pisane w asemblerze i najczęściej wykorzystywały przerwania BIOS-u lub systemu DOS. Podobnie zachowywały się sterowniki .DRV stworzone dla Microsoft Windows 3.x – graficznej nakładki na system DOS. Wprowadzony przez firmę Intel mikroprocesor 80286 udostępniał dwa tryby pracy: rzeczywisty i chroniony. W trybie chronionym mikroprocesor zapewniał 4 poziomy ochrony, od poziomu 0 - najbardziej uprzywilejowanego do poziomu 3. Uprawnienia dla poziomu 0 mają komponenty jądra systemu, natomiast uprawnienia dla poziomu 3 mają aplikacje użytkownika i wydzielona grupa elementów systemu. Zaimplementowany tryb chroniony wykorzystywany był przez system Windows, w którym obsługę urządzeń przejęły sterowniki .DRV trybu chronionego pracujące z poziomem 0.

Zapotrzebowanie na system wielozadaniowy i pojawienie się na rynku procesora 80386, który wzbogacony został o kolejny trzeci tryb pracy „wirtualny 8086", zmusiło Microsoft do stworzenia systemu operacyjnego działającego na zasadzie maszyny wirtualnej. Każda aplikacja DOS otrzymywała własną platformę 8086 z pełnym zestawem zasobów sprzętowych. Aby jednak zapobiec konfliktom dostępu do tych zasobów stworzono mechanizm wirtualnych sterowników urządzeń (pliki .VXD). Sterownik stwarzał aplikacji wirtualny interfejs urządzenia i dbał o to, aby nie dochodziło do równoczesnego dostępu do zasobu przez więcej niż jeden program. Kolejnym krokiem w dziedzinie sterowników urządzeń, było wprowadzenie przez Microsoft standardu WDM (*ang. Windows Driver Model*). Sterowniki WDM wprowadzono w systemie Windows 98 i stosowane są w kolejnych wersjach Windows.

Standard WDM zaimplementowano w:

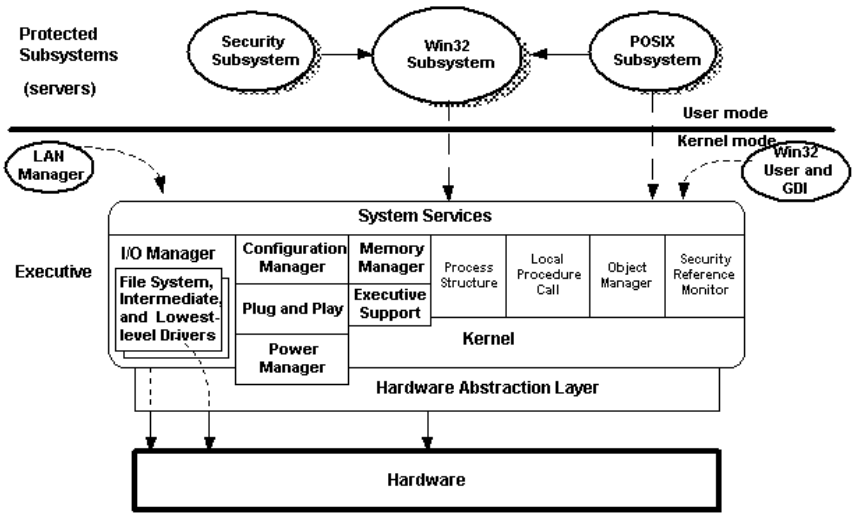
- Systemach Windows 98 i Windows Milenium – skierowanych dla użytkowników indywidualnych (określane dalej jako Windows 98/Me).
- Systemach Windows 2000, Windows XP i Windows 2003 Server – skierowanych do zastosowań sieciowych (określane dalej jako Windows 2000/XP).

2. Obsługa wejścia-wyjścia w systemach Windows 98/Me i Windows 2000/XP

Wspomniane powyżej dwie kategorie systemów Windows posiadają diametralnie różne architektury, główną zaletą sterowników WDM jest to, że pracują na obu tych platformach.

Środowisko systemowe Windows 2000/XP złożone jest z komponentów pracujących w jednym z dwóch trybów: użytkownika lub jądra (patrz rysunek 1). Tryb użytkownika jest trybem, w którym pracujące aplikacje, z przyczyn bezpieczeństwa, nie mają bezpośredniego dostępu do zasobów systemu, lecz

korzystają z wywołań funkcji systemowych. Jądro zarządza wszystkimi zasobami, sprawując kontrolę nad wykorzystywaniem ich przez aplikację i bezpośrednio nimi zarządzając. Podział na takie tryby wprowadził już Windows 3.0, jednak nie wprowadzono w nim takich systemów ochrony, jakie występują w obecnych produktach Microsoft.



Rys. 1. Architektura systemów Windows 2000/XP

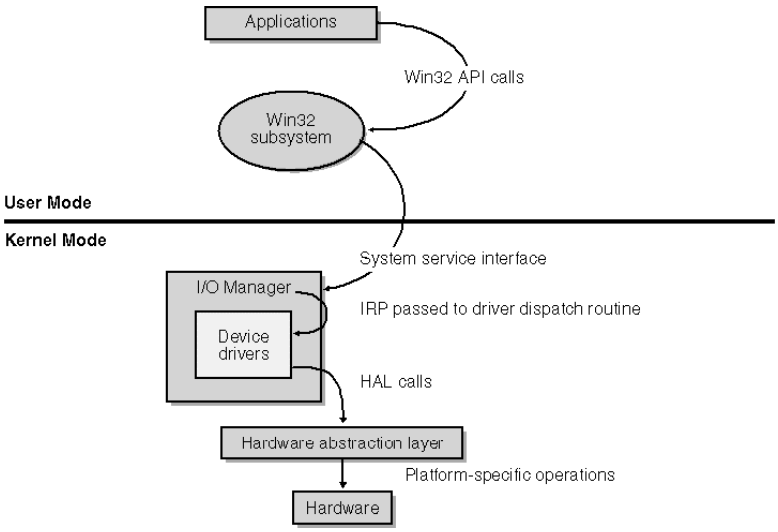
System operacyjny Windows 2000/XP zawiera komponenty trybu jądra, które są od siebie odizolowane programowo i funkcjonalnie. Sterowniki urządzeń wejścia-wyjścia współpracują przede wszystkim z takimi komponentami jak:

- zarządca wejścia-wyjścia (*ang. Input/Output Manager*);
- mechanizm Plug & Play (*ang. Plug & Play Manager*);
- mechanizm zarządzania energii (*ang. Power Manager*);
- abstrakcyjna warstwa sprzętu (*ang. HAL – Hardware Abstraction Layer*).

Komponenty te, szczególnie HAL pośredniczą przy obsłudze żądań wejścia-wyjścia co schematycznie przedstawiono na rysunku 2.

Oprogramowanie użytkowe pracuje w trybie użytkownika. Program, który żąda dostępu do zasobu sprzętowego, używa interfejsu programowego API (*ang. Application Programming Interface*). Interfejs ten udostępniany jest przez biblioteki jądra systemu takie, jak np. KERNEL32.DLL. Funkcjonalnie operacje wejścia-wyjścia wykonuje komponent zwany zarządcą wejścia-wyjścia (*ang. Input/Output Manager*), przy czym nie jest to pojedynczy wykonywalny moduł

systemu, a zbiór bibliotek systemowych odpowiedzialnych za usługi systemowe typu operacje wejścia-wyjścia.



Rys. 2. Skrócony diagram mechanizmu komunikacji aplikacji z urządzeniami I/O

W trybie jądra pracują również systemowe funkcje API stosowane do komunikacji z urządzeniami zewnętrznymi. W pierwszej fazie obsługi żądania wejścia-wyjścia sprawdzane są parametry, z jakimi funkcja została wywołana. Jeżeli wartości te ingerują w chronione obszary systemowe, funkcja nie jest wykonywana. W drugiej fazie tworzone są struktury danych zwane pakietami żądań wejścia-wyjścia (*ang. IRP – I/O Request Packet*), które przesyłane są do sterownika urządzenia. W zależności od rodzaju wykonywanej operacji, sterownik zwraca do systemu pakiet IRP o zawartości odpowiedniej rodzajowi wykonanej operacji. Może to być przykładowo:

- komunikat PENDING – informujący o tym, że zadanie jest jeszcze wykonywane i system operacyjny zostanie powiadomiony o jego zakończeniu;
- komunikat SUCCESS – informujący o tym, że zadanie zakończyło się powodzeniem;
- dane, których czytania dotyczyło polecenie wejścia-wyjścia;
- liczba bajtów danych wysłanych do urządzenia, których dotyczyło polecenie wejścia-wyjścia.

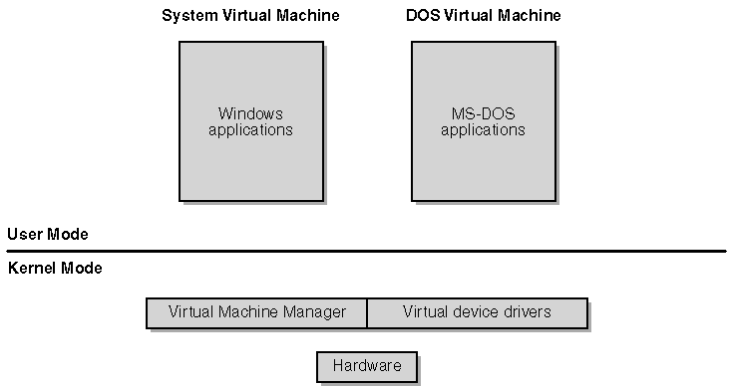
Sterownik wykonuje polecenie zawarte w odebranym pakiecie IRP, może to być zapis lub odczyt określonych portów wejścia-wyjścia lub też odwzorowanych w pamięć operacyjną rejestrów urządzenia zewnętrznego.

Sterowniki działają w trybie jądra i wykorzystują funkcje abstrakcyjnej warstwy sprzętu HAL do komunikacji ze sprzętem.

Po zakończeniu operacji wejścia-wyjścia sterownik, poprzez wywołanie odpowiednich funkcji jądra, uzupełnia danymi pakiet IRP. Następnie system przekazuje dane zawarte w IRP do aplikacji, pozwalając na dalsze jej wykonywanie.

Współpraca z urządzeniami wejścia-wyjścia pod kontrolą systemu Windows 98/Me

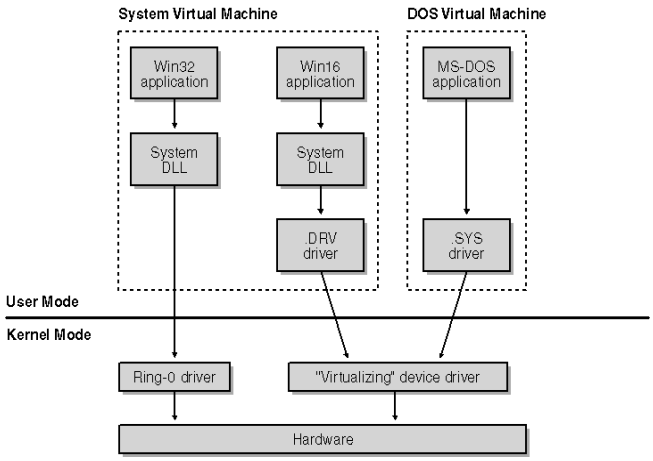
Jądro systemu operacyjnego zwane jest zarządcą maszyny wirtualnej VMM (*ang. Virtual Machine Manager*), co wynika z jego funkcji w zakresie przydziału zasobów sprzętowych komputera dla aplikacji, poprzez utworzenie maszyny wirtualnej. Takie rozwiązanie daje możliwość przydziału zasobów sprzętowych wielu aplikacjom. Architektura VMM stosowana w nowych generacjach systemów operacyjnych Microsoft jest tą samą, którą wprowadził Windows 3.0 (patrz rysunek 3), z dodatkowymi komponentami odpowiedzialnymi za obsługę nowszych urządzeń i 32-bitowych aplikacji (patrz rysunek 4).



Rys. 3. Architektura systemu Windows 98/Me

Mechanizmy komunikacji systemu Windows 98/Me z urządzeniami wejścia-wyjścia przedstawia rysunek 4. Dla aplikacji 32-bitowych żądania wejścia-wyjścia obsługiwane są przez funkcje API udostępniane poprzez biblioteki DLL takie jak np. kernel32.dll. W przeciwieństwie do systemów Windows 2000/XP nie istnieje tu jednak jeden konkretny mechanizm komunikacji ze sterownikami. Każdy rodzaj urządzenia zewnętrznego komunikuje się z systemem

wykorzystując specyficzne dla niego metody, z wyjątkiem urządzeń obsługiwanych przez sterowniki WDM, których współpraca opiera się o standard pakietów IRP.



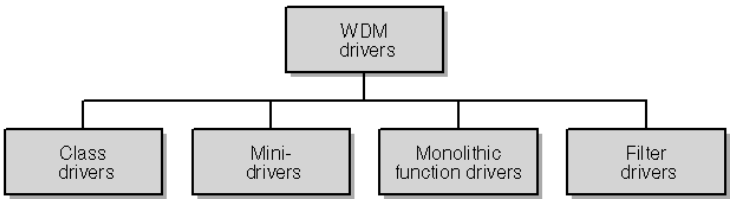
Rys. 4. Mechanizmy komunikacji systemu Windows 98/Me z urządzeniami wejścia-wyjścia

Na rysunku 4 kolumna środkowa przedstawia mechanizm komunikacji aplikacji Windows 16-bitowych, a prawa – aplikacji DOS. W obu przypadkach programy zwracają się z żądaniami bezpośrednio, czy też pośrednio, do sterownika pracującego w trybie użytkownika. Nad dostępem do zasobów ochronę sprawuje wirtualny sterownik urządzenia pracujący w trybie jądra. Ma on za zadanie umożliwić wielu maszynom wirtualnym korzystanie bez konfliktów ze wspólnych zasobów.

Operacje wejścia-wyjścia trybu jądra w Windows 2000/XP wykonywane są w oparciu o standardową strukturę IRP, Windows 98/Me korzysta z różnych mechanizmów (np. architektura warstwowa z przepływem pakietów, pakiety IRP i inne). W przypadku używania sterowników WDM, system korzysta z modułu NTKERN.VXD, w którym zaimplementowano funkcje w części pokrywające się ze stosowanymi w systemach Windows 2000/XP. Zadaniem tego modułu jest tworzenie pakietów IRP i wysyłanie ich do sterowników WDM obsługujących urządzenia. Cały proces realizowany jest tak jak to zostało wcześniej omówione. Sterowniki urządzeń, teoretycznie, nie są w stanie określić, w jakim środowisku działają. W praktyce jednak występują sterowniki WDM, które są niekompatybilne, tzn. skierowane albo dla Windows 98/Me albo 2000/XP.

3. Sterowniki WDM

Sterowniki WDM są obecnie najczęściej stosowanymi sterownikami ze względu na ich uniwersalność, przejawiającą się możliwością pracy zarówno pod kontrolą systemu Windows 98/Me oraz 2000/XP, jak również ze względu na możliwość obsługi mechanizmów Plug&Play i Power Management. Podział funkcjonalny przedstawiony jest na rysunku 5.



Rys. 5. Rodzaje sterowników WDM

Sterowniki grup urządzeń (*ang. Class Drivers*) obsługują urządzenia należące do określonej grupy funkcjonalnej. Są sterownikami wyższego poziomu, które świadczą usługi systemowe niezależnie od zastosowanego sprzętu. Aby obsłużyć urządzenia konkretnych producentów i konkretnych modeli, współpracują często ze sterownikami niskopoziomowymi dostarczonymi przez producenta (tzw. mini-sterowniki). Przykładami sterowników grup urządzeń są sterowniki klawiatur, myszy.

Mini-Sterowniki (*ang. Mini-Drivers*) to sterowniki niskopoziomowe, które dostarczają standardowy interfejs wejścia-wyjścia sprzętu dla sterowników grup urządzeń, jak również zaawansowane procedury dla oprogramowania dedykowanego dla konkretnych modeli sprzętu.

Monolityczne sterowniki funkcjonalne (*ang. Monolithic WDM Function Drivers*) są to sterowniki obsługujące w pełni określone urządzenie, nie wymagające dodatkowych komponentów programowych. Obsługują urządzenia fizyczne, jak również świadczą usługi bezpośrednio dla systemu operacyjnego.

Sterowniki filtrujące (*ang. Filter Drivers*) są sterownikami, które regulują przepływ sterowania pomiędzy systemem a urządzeniem zewnętrznym. Ich działanie polega na:

- przekształcaniu żądań systemowych;
- ukierunkowaniu sterowania pod kątem konkretnego modelu urządzenia.

4. Tworzenie sterowników przy użyciu DDK

Proces budowy sterowników wymaga platformy programowej w postaci:

- Microsoft Visual C++;
- DDK – Driver Development Kit.

Microsoft Visual C++ pełni funkcje kompilatora i edytora plików źródłowych sterownika, natomiast DDK jest zestawem oprogramowania zawierającym:

- narzędzia do debugowania sterowników;
- narzędzia do testowania sterowników;
- pliki konfiguracyjne systemu niezbędne przy pracy ze sterownikami;
- pliki nagłówkowe funkcji i struktur systemowych;
- pliki biblioteczne funkcji systemowych;
- narzędzia do manipulowania elementami jądra systemu.

Środowisko daje możliwość kompilacji sterownika do jednej z dwóch postaci:

- *Checked* - sterownik zawiera dodatkowe informacje pomocne w trakcie jego testowania i debugowania. Pozwalają one na śledzenie pracy sterownika i weryfikację poprawności jego działania.
- *Free* – jest to wersja sterownika zawierająca minimalną ilość kodu, która zapewnia jego poprawne działanie.

Systemy Windows 2000/XP występują w dwóch wersjach:

- *Free build* jest wersją, z jaką pracuje użytkownik końcowy. System i sterowniki są w pełni zoptymalizowane pod kątem wydajności i zużycia pamięci.
- *Checked build* jest wersją stosowaną przy badaniu działania systemu i jego komponentów. Zawiera ona mechanizmy wykrywania błędów systemu, weryfikowania parametrów wywołań systemowych i debugowania. Wersja ta pozwala na odizolowanie i śledzenie problemów występujących w trakcie pracy systemu. Ze względu na dodatkowe czynności wykonywane w trakcie działania systemu i dodatkowe informacje w plikach wykonywalnych, wersja ta jest wolniejsza od wersji „free”.

Proces budowy sterownika jest procesem wieloetapowym i wymaga zwykle realizacji następujących kroków:

- napisania kodu i dołączenia do niego fragmentów informacyjnych dla debugera;
- testowania i debugowania wersji ‘checked’ sterownika przy użyciu systemu Windows w trybie ‘checked’;

- testowania i debugowania wersji ‘free’ sterownika przy użyciu systemu Windows w trybie ‘free’;
- optymalizacji wersji ‘free’ sterownika pod kątem wydajności;
- końcowego testowania obu wersji sterowników.

Przy tworzeniu sterownika można zastosować jedną z dwóch dróg realizacji tego procesu:

- utworzyć sterownik na bazie kodu wzorcowego, poprzez jego modyfikację i dopasowanie do założeń projektowych. Środowisko DDK zawiera zestaw przykładowych kodów źródłowych sterowników dla najczęściej stosowanych urządzeń;
- użycie programu wspomagającego tworzenie sterowników. Programy wspomagające pisanie sterowników umożliwiają automatyczne wygenerowanie szkieletu sterownika. Tworzą one podstawowe elementy kodu źródłowego, dołączając odpowiednie pliki nagłówkowe, deklarując podstawowe funkcje i struktury. W przypadku niektórych narzędzi można również generować pliki informacyjne .INF wykorzystywane w procesie instalacji sterownika w systemie.

Tworzenie sterownika za pomocą DDK

Podstawowym narzędziem do kompilacji i budowania pliku binarnego sterownika jest program BUILD, powiązany bezpośrednio z Microsoft Visual C++. Jest to program pracujący w trybie tekstowym i jest uruchamiany w konsoli pracującej w jednym z dwóch trybów: *checked* lub *free*. Dostępny jest w Menu Start po uprzednim zainstalowaniu pakietu DDK. Korzysta z pliku „SETENV.BAT” zawierającego ustawienia wymaganych parametrów systemu operacyjnego (np. zmiennych systemowych, ścieżek dostępu).

Uruchomienie tego narzędzia realizowane jest poprzez wpisanie w linii poleceń:

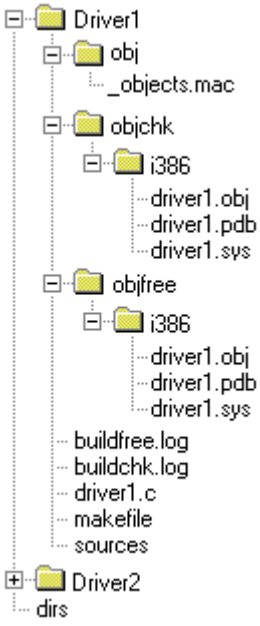
- **Build -?**: co powoduje wyświetlenie składni wywołania tego polecenia;
- **Build –cZ**: co powoduje uruchomienie procesu tworzenia pliku binarnego sterownika z jednoczesnym usunięciem wcześniejszych rezultatów kompilacji.

Stosowanie narzędzia BUILD wymaga wcześniejszego przygotowania następujących plików konfiguracyjnych:

- *sources* – zawiera nazwy wszystkich plików, w danym katalogu, z kodem źródłowym tworzonego sterownika. Zawiera on również zestaw makr określających cechy tworzonego sterownika. Spis dostępnych makr zawarty jest w dokumentacji DDK;

- *dirs* – zawiera nazwy podkatalogów w danym katalogu, w których znajdują się pliki źródłowe;
- *makefile* – wymagany jest przez środowisko Microsoft Visual C++ w celu określenia zależności między plikami źródłowymi, ustawienia odpowiednich flag dla kompilatora i linkera.

Struktura przykładowego projektu sterownika przedstawiona jest na rysunku 6.



Rys. 6. Przykładowa struktura projektu sterownika

Zawartość przykładowego pliku *sources:*

```
TARGETNAME=packet
TARGETPATH=obj
TARGETTYPE=DRIVER

TARGETLIBS=$(DDK_LIB_PATH)\ndis.lib
C_DEFINES=$(C_DEFINES) -DNDIS50

INCLUDES=..\..\inc;..\inc;
SOURCES=packet.c  \
```

```
        read.c    \
        write.c   \
```

Zawartość przykładowego pliku *dirs*

```
dirs= Driver1   \
      Driver2   \
      Driver3
```

Aby ustawić katalog domyślny projektu zawierającego pliki z kodem źródłowym należy użyć polecenia zmiany katalogu: 'CD'. Polecenie BUILD dokona kompilacji i linkowania sterownika a pliki wynikowe umieści w podkatalogu \objfre lub \objchk w zależności od wybranej opcji środowiska (free lub checked).

Wynikiem wywołania polecenia BUILD, oprócz pliku sterownika są również dodatkowe pliki informacyjne:

- *build.log* – zawierający informacje o wynikach kompilacji i linkowania;
- *build.wrn* – zawierający listę ostrzeżeń wygenerowanych w trakcie budowania sterownika;
- *build.err* – zawierający listę błędów wygenerowanych w trakcie budowania sterownika.

5. Narzędzia wspomagające tworzenie sterowników

W celu uproszczenia procesu tworzenia sterowników opracowano wiele narzędzi przeznaczonych do generowania szkieletu kodu źródłowego sterownika, zawierającego deklaracje i częściowo definicje podstawowych procedur i struktur danych. Poniżej przedstawiono krótką charakterystykę tych narzędzi, do których Autor artykułu miał dostęp.

5.1. KernelDriver

KernelDriver® jest produktem firmy Jungo Software Technologies. Stworzony został w celu przyspieszenia tworzenia i optymalizowania kodu sterownika. Skierowany jest na tworzenie sterowników dla standardowych kategorii urządzeń zewnętrznych, które wymagają bezpośredniego dostępu do sprzętu i mają działać w trybie jądra.

Pakiet ma bardzo przydatne komponenty, które pozawalają automatycznie wykrywać podłączone do komputera urządzenia, komunikować się z poziomu

programu z portami wejścia-wyjścia, rejestrami urządzeń, fragmentami pamięci przeznaczonymi dla urządzeń.

Zalety programu KernelDriver to:

- Bezpośredni dostęp do sprzętu — Program pozwala na testowanie i diagnozowanie urządzeń zewnętrznych poprzez przyjazny interfejs graficzny aplikacji pracującej w trybie użytkownika.
- Automatyczne generowanie kodu — Generowany jest szkielet sterownika z elementami już skierowanymi na żądane urządzenie.
- Środowisko okienkowe — KernelDriver jest aplikacją okienkową, dzięki czemu jest bardzo wygodny w użyciu.
- Obsługa wielu urządzeń — KernelDriver obsługuje niezależnie od producenta urządzenia oparte o magistrale PCI/CardBus/ISA/ISAPnP/EISA/CompactPCI i USB.
- Obsługa platform wieloprocesorowych — KernelDriver obsługuje maszyny wieloprocesorowe z wielokrotnymi magistralami PCI.
- Kompatybilność binarna — Sterowniki wygenerowane przez KernelDriver są binarnie kompatybilne w systemach Windows 98/Me/NT/2000/XP/Server 2003.
- Kompatybilność źródłowa — Sterowniki wygenerowane przez KernelDriver są kompatybilne na poziomie kodu źródłowego w systemach Windows 98/Me/NT/2000/XP/ Server 2003.
- Standardowe środowisko programistyczne — Kompilacja i budowanie sterownika może odbywać się w oparciu o środowisko Microsoft Studio, gcc lub każdy inny 32-bitowy kompilator.
- Wsparcie technologii 64-bitowej — KernelDriver umożliwia korzystanie z zalet 64-bitowej magistrali PCI, transferu 64-bitowego na platformie x86 pracującego z systemem operacyjnym 32-bitowym.
- Dynamicznie wczytywane sterowniki — KernelDriver posiada narzędzia do dynamicznego wczytywania i usuwania z systemu sterowników. Cecha bardzo przydatna przy debugowaniu sterowników.
- Dokładna dokumentacja — KernelDriver posiada dokładną dokumentację procesu budowania sterowników, obsługi programu.
- Przykłady w C — Do pakiety KernelDriver dołączone są kody źródłowe przykładowych sterowników, z których można skorzystać pracując nad własnym kodem.

Wersja 30-o dniowa programu KernelDriver dostępna jest na stronie WWW producenta: www.jungo.com

5.2. WinDriver

Windriver® jest kolejnym produktem firmy Jungo Software Technologies. Jest to oprogramowanie, które integruje się z pakietem KernelDriver

i skierowane jest do twórców sterowników i aplikacji korzystających z zasobów sprzętowych komputera.

Największymi zaletami pakietu WinDriver są:

- możliwość stworzenia sterownika urządzenia pracującego w trybie użytkownika, który posiada zdolność do bezpośredniego komunikowania się ze sprzętem poprzez specjalny sterownik jądra windriver.sys, który instaluje się automatycznie z pakietem WinDriver lub KernelDriver;
- możliwość napisania aplikacji, która dzięki wygenerowanemu sterownikowi, może z poziomu trybu użytkownika oddziaływać bezpośrednio na zasoby sprzętowe;
- końcowa faza pracy WinDriver tworzy obok plików źródłowych sterownika, wszystkie pliki projektu dla Microsoft Visual Studio.

Wymagane jest, aby sterowniki wygenerowane przez Windriver używane były przez dedykowane im aplikacje.

Wersja 30-o dniowa programu WinDriver dostępna jest na stronie WWW producenta: www.jungo.com

5.3. Driver Wizard

Driver Wizard autorstwa Dudiego Avramova jest kolejnym narzędziem do generowania szkieletu sterownika, ustępującym jednak pod względem funkcjonalności dwóm wcześniejszym narzędziom firmy Jungo. Driver Wizard jest szablonem projektu dla Microsoft Visual Studio, w którym zadeklarowane są podstawowe funkcje i struktury sterownika. Do funkcji deklarowanych z poziomu tego narzędzia należą:

- „*DriverEntry*” – tzw. funkcja wejścia do sterownika, używana przy inicjalizowaniu sterownika;
- „*DrvDispatch*” – funkcja obsługi pakietów IRP przesyłanych do sterownika;
- „*DrvUnload*” – funkcja zwalniania zasobów zajętych przez sterownik, która uruchamiana jest w trakcie usuwania sterownika z pamięci.

Do pobrania dostępny jest gotowy plik DriverWizard.awx, który należy skopiować do katalogu pakietu Microsoft Visual Studio: ..\Microsoft Visual Studio\Common\MSDev98\Template

Po uruchomieniu środowiska należy utworzyć nowy projekt i wybrać z dostępnego menu szablonów opcję “Driver AppWizard”.

Driver Wizard dostępny jest w Internecie pod adresem:

http://www.codeproject.com/useritems/driverwizard.asp.

5.4. WDM Wizard

WDM Wizard autorstwa Waltera Oneya jest generatorem szkieletu sterownika, działającym w oparciu o środowisko Microsoft Visual Studio. Narzędzie to działa na zasadzie kreatora projektu. Przy tworzeniu nowego projektu w Visual Studio należy wybrać kategorię „WDM Driver Wizard". Praca kreatora składa się z kroków, w których podaje się cechy jakie powinien posiadać sterownik.

Krok 1: wymaga określenia rodzaju tworzonego sterownika (funkcjonalny, filtrujący itp.), jak również opcji dodatkowych takich jak lokalizacja instalacji DDK.

Krok 2: wymaga określenia rodzaju komunikacji urządzenia z systemem (przerwania, obszar wejścia-wyjścia, mapowana pamięć), jak również parametrów komunikacji DMA o ile taka będzie używana.

Krok 3: określa własności urządzenia dla którego tworzony jest sterownik. (Klasa urządzenia, Identyfikator urządzenia, identyfikator producenta, nazwa urządzenia, opis urządzenia).

Po zakończeniu kreatora tworzony jest nowy projekt w Visual Studio, w którym zadeklarowane są podstawowe struktury i funkcje sterownika. Wynik działania tego kreatora jest podobny do wyników narzędzia Driver Wizard, z tą różnicą, że dzięki podaniu wiadomości dodatkowych, deklarowana jest większa ilość funkcji i struktur.

6. Uruchamianie i testowanie sterowników

Sterowniki urządzeń pracujące w trybie jądra posiadają dostęp do wszystkich zasobów komputera, tak więc niepoprawna ich praca może zachwiać pracę systemu i może doprowadzić do jego zawieszenia. Dla utworzonych sterowników przeprowadza się wnikliwe etapy testowania i debugowania w celu wyeliminowania błędów.

Procedura testowanie sterownika wymaga jego wcześniejszego załadownia do pamięci za pomocą jednej z dwóch metod:

- za pomocą narzędzi Windows takich jak menadżer urządzeń, poprzez kreatora poszukiwania nowego sprzętu lub też aktualizację sterownika, jeżeli sprzęt został przez system odnaleziony i zainstalowano sterownik zawarty w systemie;
- za pomocą WDREG.EXE – narzędzia z pakietów KernelDriver i WinDriver służącego do dynamicznego ładowania sterowników w systemie Windows.

Narzędziem do testowania sterowników jest aplikacja Driver Verifier udostępniana z razem z systemem Windows, jak również środowiskiem DDK. Wersja systemowa weryfikatora znajduje się w katalogu głównym systemu ..\System32\Verifier.exe. Wersja DDK dostępna poprzez menu start: Menu Start\Programy\ Development Kits\Windows DDK\Driver Verifier.

Weryfikator sterowników to mechanizm służący do wyszukiwania i wyodrębniania często występujących błędów w sterownikach urządzeń lub innym kodzie systemowym trybu jądra. Proces testowania sterownika wymaga uruchomienia weryfikatora i w zakładce **Ustawienia** wybrania żądanego sterownika (sterowników), który jest przedmiotem badania. Dodatkowo ustala się kryteria sprawdzania sterownika. Dostępne są następujące opcje sprawdzania:

- **pula specjalna** – sprawdzane są odwołania sterownika do obszarów pamięci, jeżeli odwołanie wykracza poza przydzielony obszar, błąd jest zanotowany przez weryfikator. Opcja ta sprawdza również poprawność przydzielania i zwalniania przez sterownik pamięci;
- **śledzenie puli** – test polegający na monitorowaniu wykorzystania pamięci sterownika, pozawala na wykrycie tzw. „wycieków" pamięci (*ang. memory leaks*) – błąd nie zwolnienia pamięci, której sterownik już nie używa;
- **wymuszanie sprawdzania poziomu żądania przerwania** (ang. IRQL – Interrupt Request Level**)** – test polegający na wykrywaniu sytuacji, gdy sterownik uzyskuje dostęp do danych lub kodu, które mogą być stronicowane, gdy procesor jest na podwyższonym poziomie IRQL;
- **symulacja niedoboru zasobów** – test polegający na sprawdzaniu czy próby przydziału pamięci wykonywane przez sterownik nie kończą się niepowodzeniem. Sytuacja ta może zdarzyć się w przypadku niskiego stanu zasobów.

Weryfikator sterowników w trakcie testowania zlicza napotkane błędy i przedstawia je w zakładce **Liczniki Globalne**.

Aby uruchomić sprawdzanie sterownika, należy zaznaczyć go w zakładce **Ustawienia**, potwierdzić zmiany i powtórnie uruchomić system, ponieważ zmiany wprowadzone zostaną w systemie po ponownym jego załadowaniu.

7. Debugowanie sterowników

Debugowanie to proces śledzenia wykonującego się oprogramowania w celu odnalezienia błędów w kodzie.

Proces debugowania składa się z dwóch etapów: umieszczenia informacji dla debugera w kodzie sterownika oraz użycia jednego z dostępnych narzędzi debugujących.

- NTSD,
- CDB,
- KD,
- WinDbg.

NTSD - *NT Symbolic Debugger* to program działający w trybie tekstowym, przeznaczony do debugowania aplikacji i sterowników pracujących w trybie użytkownika znajdujący się w katalogu system32 systemu Windows. Jest bardzo dobrym narzędziem, jak również łatwym w konfiguracji. W momencie wystąpienia błędu badanej aplikacji, NTSD ma możliwość śledzenia stosu, parametrów wywołań funkcji. NTSD daje możliwość wyświetlania i wykonywania kodu, ustawiania punktów kontrolnych (*ang. breakpoints*), kontrolowania i modyfikowania zawartości pamięci. Dodatkowo przy jego pomocy można odwoływać się do pamięci poprzez nazwy symboliczne lub adresy, dzięki czemu ułatwia wyszukiwanie określonych fragmentów kodu. Obsługuje aplikacje wielowątkowe oraz ma możliwość zapisu i odczytu pamięci stronicowanej jak i nie stronicowanej.
Wadą tego debugera jest brak możliwości analizowania plików zrzutu pamięci.

CDB - *Console Debugger* to odmiana NTSD, która w przeciwieństwie do NTSD działa w oknie konsoli, w którym została wywołana (NTSD tworzy własne okno), pozwala na debugowanie zdalne.

KD - *Kernel Debugger* to program działający w trybie tekstowym, który pozwala na dogłębną analizę działania systemu w trybie jądra. KD używany jest do monitorowania zachowania się programów i sterowników, jak również systemu operacyjnego. Narzędzie to wymaga do poprawnej realizacji procesu debugowania dwóch komputerów, co przedstawiono schematycznie na rysunku 7. Jeden z nich pełni rolę maszyny docelowej pracującej pod kontrolą debugowanego systemu, drugi natomiast pełni rolę maszyny sterującej (*ang. host machine*), która monitoruje działanie pierwszej. KD nie jest użyteczna w przypadku debugowania aplikacji trybu użytkownika, ponieważ nie ma możliwości ustawiania w tym trybie punktów kontrolnych.

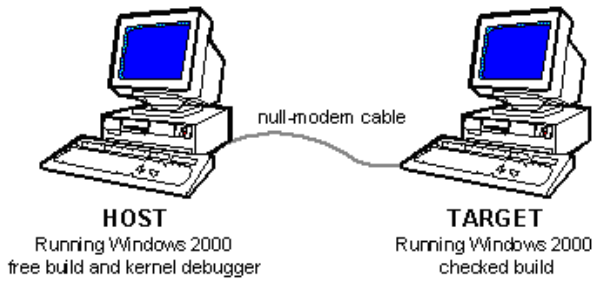
WinDbg- *Windows Debugger* to okienkowe narzędzie do debugowania kodu działającego w trybie użytkownika i jądra (aplikacje, sterowniki, usługi systemowe). Korzysta ono z formatu symboli debugera Microsoft Visual Studio, ma dostęp do kodu wszystkich funkcji i zmiennych globalnych, które zadeklarowane zostały w modułach skompilowanych z plikami symboli. Narzędzie to ma możliwość przeglądania wykonywanego kodu, ustawiania punktów kontrolnych, monitorowania zmiennych, śledzenia stosu i pamięci. Oprócz funkcji

dostępnych poprzez standardowe menu programu, WinDbg realizuje polecenia z linii komend. Analogicznie jak KD, wymaga stosowania dwóch komputerów do debugowania kodu wykonywanego w trybie jądra.
WinDbg jest bardziej złożonym narzędziem niż KD czy NTSD i przez to wolniejszym, ale dzięki temu wysoce funkcjonalnym.

Konfiguracja sprzętowa wymagana przy debugowaniu

Debugowanie sterownika pracującego w trybie jądra wymaga komputera docelowego i komputera hosta. Komputer docelowy używany jest jako maszyna na której uruchomiony jest sterownik, komputer host ma uruchomiony debuger.

Rysunek 7 przedstawia podstawową konfigurację sprzętu wymaganą do lokalnego debugowania sterownika.



Rys. 7. Konfiguracja sprzętowa do lokalnego debugowania sterowników

Ustawienie połączenia między komputerami.

Komputer docelowy łączony jest z hostem poprzez port szeregowy, połączenie typu Null-modem. Domyślnie używanym portem na obu maszynach jest COM2 i prędkość 19200b/s.

Ustawienia portu i prędkości:

KD - za parametry transmisji odpowiedzialne są zmienne środowiskowe:
port _NT_DEBUG_PORT=com[1|2|...],
prędkość _NT_DEBUG_BAUD_RATE=baud rate

NTSD - parametry transmisji podaje się w linii komend:
port port = <COM PORT>
prędkość baud = <BAUD RATE>

WinDbg - parametry transmisji wybiera się w menu View/Options, zakładka Transport Layer.

Konfigurowanie maszyny docelowej

Możliwość zainicjowania trybu debugowania jądra może być zrealizowana poprzez dodanie linii /debug w pliku boot.ini. Możliwe opcje uruchomienia systemu przedstawiono w tabeli 1.

Tab. 1. Dostępne opcje uruchamiania systemu Windows 2000/XP

Opcja	Opis
`/nodebug`	Wyłącza tryb debugowania jądra. Jest to ustawienie domyślne.
`/debug`	Powoduje wczytanie przy ładowaniu systemu debugera, który działa w systemie przez cały czas pracy. Dzięki temu możliwe jest podłączenie się to systemu z zewnątrz.
`/debugport=`*`Port`*	Ustawia numer portu szeregowego używanego przy debugowaniu, domyślnie jest to COM1.
`/crashdebug`	Powoduje wczytanie debugera w trakcie ładowania systemu i automatyczne przerzucenie go do pliku wymiany. W wyniku tego nie można połączyć się z debugerem, dopóki system nie zostanie przestawiony w tryb uśpienia albo zatrzymany.
`/baudrate=`*`BaudRate`*	Ustala prędkość transmisji między komputerem docelowym i hostem, wyraża się w bitach/sekundę.
`/kernel`	Opcja ta pozwala na wybranie innego niż standardowy pliku jądra. Systemowe jądro znajduje się w katalogu *c:\windows\system32*. Opcji tej używa się w trakcie manipulowania kopią jądra, bez ryzyka uszkodzenia domyślnej wersji.
`/hal`	Opcja ta pozwala na wybranie innego niż standardowy pliku obsługi warstwy HAL. Dzięki temu można testować kopie tego pliku bez ryzyka uszkodzenia systemu operacyjnego.
`/maxmem=`*`SizeInMB`*	Określa wielkość pamięci dostępnej dla systemu. Dzięki temu można ograniczyć ilość pamięci w systemie komputerowym.
`/sos`	Wymusza wyświetlanie na ekranie monitora nazw sterowników aktualnie wczytywanych przez system.

Zmiany w pliku boot.ini można wykonać w dowolnym edytorze. Przykład pliku boot.ini systemu Windows XP Professional, w którym ustawiono tryb debugowania i prędkość transmisji na 9600 b/s przedstawiono poniżej.

```
[boot loader]
```

```
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft
Windows XP Professional" /fastdetect /debug /baudrate=9600
```

Aby wprowadzone zmiany odniosły skutek, należy uruchomić system ponownie.

Konfiguracja komputera hosta

W przypadku używania debugera KD, należy ustalić odpowiednie zmienne środowiskowe, których część używana jest również przez NTSD. Tabela 2 zawiera zbiór dostępnych zmiennych środowiskowych, które mogą być użyteczne w trakcie debugowania sterownika.

Przykładowy zestaw zmiennych środowiskowych:

- _NT_SYMBOL_PATH=c:\windows\symbols
- _NT_ALT_SYMBOL_PATH=e:\SP\symbols
- _NT_DEBUG_PORT=com1
- _NT_DEBUG_BAUD_RATE=19200
- _NT_LOG_FILE_OPEN=c:\DEBUG.LOG

Zmiennymi, których pominąć nie można, są :

- _NT_DEBUG_PORT;
- _NT_SYMBOL_PATH.

Komputer host musi posiadać również zainstalowany zestaw symboli. Pliki symboli tworzone są w trakcie linkowania sterowników, aplikacji, systemów operacyjnych i zawierają dane przydatne przy ich debugowaniu.

Przykładowymi informacjami przechowywanymi w plikach symboli są:

- nazwy zmiennych globalnych;
- nazwy funkcji i adresy ich punktów wejścia;
- numery linii kodu źródłowego.

Nie są one potrzebne w trakcie normalnego uruchamiania plików binarnych, dlatego producent nie dołącza ich standardowo do systemu operacyjnego.

Szczegóły na temat pozyskiwania plików symboli przedstawione są w Internecie pod adresem:

http://www.microsoft.com/whdc/ddk/debugging/symbols.mspx.

Pliki symboli tworzone są również w trakcie linkowania własnoręcznie tworzonych sterowników. Microsoft Visual Studio tworzy pliki symboli w standardzie Windows, w plikach .pdb i dbg.

Tab. 2. Zmienne środowiskowe używane w trakcie debugowania systemu

Zmienna	Znaczenie
_NT_DEBUG_PORT=*port*	Określa numer portu szeregowego, przez który dołączony jest komputer docelowy. Wartość domyślana to COM1.
_NT_DEBUG_BAUD_RATE=baud	Określa prędkość transmisji. Wartość domyślna to 19200.
_NT_SYMBOL_PATH=*path[;path…]*	Nazwa katalogu, w którym zainstalowano pliki symboli.
_NT_ALT_SYMBOL_PATH=*path*	Nazwa katalogu w którym znajduje się alternatywny zestaw symboli. Używane do przechowywania własnych plików symboli.
_NT_DEBUGGER_EXTENSION_PATH =*path*	Ścieżka do katalogu w którym debuger wyszukiwał będzie bibliotek DLL.
_NT_DEBUG_LOG_FILE_OPEN =*filename*	Nazwa pliku w którym KD stworzy plik raportu z przeprowadzonych czynności.
KDQUIET=*anything*	Debuger nie wyświetla informacji za każdym razem gdy załadowano lub usunięto z pamięci bibliotekę DLL. Debuger nie wyświetla ostrzeżenia że naciśnięte zostały kombinacje klawiszy CTRL+C czy CTRL+BREAK.
_NT_DEBUG_CACHE_SIZE=*size*	Określa maksymalny rozmiar pamięci podręcznej dla KD (w bajtach).

Literatura

[1] *Microsoft Software Developement Library*, June 2003.
[2] Solomon A. D., Russinovich E. M., *Microsoft Windows 2000 od środka*, Helion, 2003
[3] Oney W.: *Programming the Microsoft Windows Driver Model*, Microsoft Press, 2003
[4] McDowell S.: *Windows 2000 Kernel Debugging*, Precentice Hall, 2001
[5] Cant C.: *Writing Windows WDM Device Drivers*, CMPbooks, 1999.

[6] Chudzikiewicz J.: *Obsługa wejścia-wyjścia w systemie Windows NT 4.0*, Biuletyn Instytutu Automatyki i Robotyki nr 10, Wydział Cybernetyki Wojskowej Akademii Technicznej 1999.

Recenzent: dr inż. Jan Chudzikiewicz

Praca wpłynęła do redakcji: 01.12.2003r.