PCMCIA

SDK

PC Card Software Development Kit

7.00

CardWare SDK for PnP OSs

Release 7.00

Proprietary Notice and Disclaimer

APSoft.
Sonnenstrasse 26b
85622 Feldkirchen
Germany

Tel: +49 (0) 89 900 479 0
Fax: +49 (0) 89 900 479 11
Internet: http://www.tssc.de

Table of Contents

This page is intentionally blank.

CHAPTER 1

Introduction

Several Microsoft OSs provide a so-called Plug-n-Play support. In Plug-n-Play OSs a dedicated driver supports every hardware device compatible with the Microsoft PnP API. Therefore your own PC Card drivers for PnP OS must have the same architecture.

There are two kinds of PnP OSs on the market:

- Microsoft Windows 95, Microsoft Windows 98 and Microsoft Windows ME (named collectively Win9x in this manual).
- Microsoft Windows 2000 and Microsoft Windows XP (named collectively WinNT in this manual).

All Microsoft PnP OSs provide a build-in support for 16-bit and CardBus cards (however, Windows 95 and early versions of Windows 98 don’t support CardBus cards). Such support includes detection of PC Card insertion and removal, card identification and loading of an appropriate device driver. Microsoft OSs are also shipped with drivers for certain types of PC Cards such us ATA/ATAPI, but if you develop your own PC Card most likely you will have to write a special driver.

Unfortunately, WinNT DDK documentation doesn’t contain an extensive description for writing PCMCIA card drivers. This manual describes the working mechanism of PnP subsystems in different PnP OSs, structure of device drivers in a PnP environment, and specific problems related to PC Cards support.

All PnP OSs provide similar support for PnP drivers. However, developers should be aware of several differences in PC Card support from one OS to another.

For example, all Win9x OSs provide Card Services APIs based on the PC Card Standard released by the International Standardization Group PCMCIA. Despite the fact, that Microsoft only supports a relatively old version of the PC Card Standard, the Card Services API provides a rich set of functions for PC Card support. WinNT OSs, however, don’t provide the Card Services API, and the set of options for PC Card driver developers are minimal.

This page is intentionally blank

C H A P T E R 2

Installation of PCMCIA Drivers

PnP Device Installation Issues

When the user connects a new device to the bus, the driver of the bus detects the new device, enumerates it, retrieves identification information, and passes this information to the Plug & Play subsystem.Plug & Play devices normally have the following identification information:

- Device and instance ID (device node name)
- Device capabilities
- One or more hardware IDs and zero or more compatible IDs

Note: Depending on the bus type, different enumerators are used to enumerate a new device. Depending on enumerator, information about the device will be stored in different branches of the registry:

\Enum**PCI**	for PCI and CardBus cards
\Enum**PCMCIA**	for 16-bit PCMCIA cards

The Windows' Plug & Play subsystem gets identification information, then

- Searches for device ID matches in the set of available INFs
- Ranks the matched ID entries in the INFs according to signature, ID match, and DriverVer date (in that order)
- Selects the best ID match – identifies the INF containing that ID
- Uses the ID entry in that INF to install the driver referenced in the INF

Notes: Here is the difference between Windows 95/98/ME and Windows 2000/XP. <u>*First of all*</u>, Windows 95/98/ME uses a pre-built device INFs database (DRVDATA.BIN and DRVIDX.BIN in C:\Windows\Inf subdirectory), where as Windows 2000/XP uses individual pre-compiled INF files (PNF-files). To change the pre-calculated rank of device IDs, both files (DRVDATA.BIN and DRVIDX.BIN) should be deleted under Windows 95/98/ME. <u>*Secondly*</u>, in the case that two or more INF files have equal device IDs, HW wizard under Window 95/98/ME ranks INFs the following way:

- It tries to read DriverVer value from [Version] section of INF file and takes a file with latest date
- If there is no DriverVer value or they are equal, HW Wizard takes last modification date/time of the INF file
- If concurrent files have equal date/time, HW Wizard takes the one that was written to directory earlier

In all OSs digitally signed drivers always have priority over digitally unsigned drivers.

INF files for Windows 95/98/ME and Windows 2000/XP have slightly different formats. Please refer to the help of corresponding DDK ('Device Installation and INF files' chapter in 95 DDK, 'Setup, Plug & Play and Power Management' chapter in 98/ME DDK, or 'Device installation' chapter in 2000/XP DDK) to find out correct format of INF file for each OS, and list of rules according to which INF file should be written.

What is a Plug & Play ID?

A Plug & Play ID is a vendor-defined string used by Windows Plug & Play to find a driver for a device. There are three kinds of Plug & Play IDs – 'Hardware IDs', 'Instance IDs' and 'Compatible IDs'. One device can have one ore more Plug & Play IDs.

Hardware ID identifies type of device. Format of HW ID depends on the enumerator (bus): <enumerator\<enumerator-specific-device-ID>. For example: PCI\VEN_nnnn&DEV_nnnn&SUBSYS_nnnnnnnn&REV_nn or PCMCIA\<card_manufacturer_ID>

You may find more information about HW ID in DDK help: 'Device IDs' chapter in Windows 95 DDK, or in Glossary of Windows 98/ME/2000/XP DDK.

Instance ID Instance ID is a string that distinguishes a device from other devices of the same type on a machine. An instance ID is persistent across system boots.

Compatible ID has the same format as Hardware ID, but is typically more generic than Hardware ID. Normally, Compatible ID is used by the Windows Plug & Play subsystem for device identification when there is no Hardware ID match. Device can expose zero or more Compatible IDs.

Example of Compatible ID: PCI\VEN_nnnn&DEV_nnnn.

You may find more information about HW ID in DDK help: ‘Compatible IDs’ chapter in Windows 95 DDK, or in Glossary of Windows 98/ME/2000/XP DDK**.**

This page is intentionally blank.

CHAPTER 3

Hardware Key for PCMCIA Cards

Device ID Generation Method

DEVICE ID generation for 16-bit PCMCIA card with CIS

If a PCMCIA card has CIS, the PCMCIA bus driver generates the DEVICE ID based on information of the CIS tuple with code 15h (CISTPL_VERS_1). The format of DEVICE ID is:

PCMCIA\MANUFACTURER-PRODUCT-CRC(4), where:

MANUFACTURER	Manufacturer name field from CISTPL_VERS_1 tuple.
PRODUCT	Product name field from CISTPL_VERS_1 tuple.
CRC(4)	4-digits hexadecimal value, which represents CRC code of CIS. In order to calculate CRC, Microsoft OS parses first kilobyte of attribute memory. If CISTPL_END is found inside, following tuples are used for CRC calculation: CISTPL_DEVICE, CISTPL_VERS_1, CISTPL_CONFIG, CISTPL_CFTABLE_ENTRY, CISTPL_MANFID, and CISTPL_END. During the calculation, CISTPL_VERS_1 is abbreviated after two first strings (Manufacturer name and Product name). All other tuples in CIS are ignored.

The Value generated in such a way should be unique for any PCMCIA card.

DEVICE ID generation for 16-bit PCMCIA card without CIS

The PCMCIA driver doesn’t support cards without CIS, except for memory cards. If the PCMCIA driver detects that the type of card is a FLASH or SRAM, it generates the DEVICE ID. In most cases it is **MTD-0000** for SRAM and **MTD-0002** for FLASH cards.

The algorithm of the memory cards recognition in the PCMCIA bus driver is not complete, and some restrictions of this algorithm can cause problems with device support. For example, if the card is recognized incorrectly, the resource requirements list can either be empty or invalid, and the card driver will be forced to build it at the receiving IRP_MN_FILTER_RESOURCE_REQUIREMENTS (Win9x: CONFIG_FILTER) request.

DEVICE ID generation for 32-bit CardBus card

The Device ID is generated based on values read from the PCI configuration space as

PCI\VEN_<VID>&DEV_<DID>&SUBSYS_<SID>&REV_<RID>,

Where

<VID>	PCI Vendor ID (four digits, hex)
<DID>	PCI Device ID (four digits, hex)
<SID>	Subsystem Vendor and Device ID (eight digits, hex)
<RID>	Revision number (two digits, hex)

Instance ID Generation

Instance ID generation for 16-bit PCMCIA card

The Instance ID is generated as a serial number of the socket. The numeration of sockets is continuous. In case of multiple PCMCIA adapters

Adapter	Socket	Instance ID of inserted card
Adapter 1	Socket 0	0
	Socket 1	1
Adapter 2	Socket 0	2
	Socket 1	3

Instance ID generation for 32-bit PCMCIA card

The Instance ID of PCI cards is a string describing location over PCI bus graph. It includes the slot number of the card and all parent PCI-2-PCI and CardBus bridges. Format of ID is

<parent prefix string>&<device path>

<parent prefix string>	Format is unknown. Prefix string is not used under Win9x.	
<device path>	<dev slot><slotX><slot1><slot0>	
	<slot0>	Slot number of bridge on PCI bus 0 (Bridge0)
	<slot1>	Slot number of bridge on secondary bus of Bridge0
	<dev slot>	Slot number of device

For example, in case of AGP video card (Bus 1 Dev 0 Func 0) located behind AGP bridge with PCI device handle - Bus 0 Dev 1 Func 0 <device path> will be **0008**:

$00 = DevNum_{card} * 8 + FuncNum_{card} = 0 * 8 + 0$

$08 = DevNum_{bridge} * 8 + FuncNum_{bridge} = 1 * 8 + 0$

Device Key

Device key of inserted card is placed into registry at following location:

Win2K/XP:
HKLM\System\CurrentControlSet\Enum\<DEVICE ID>\<INSTANCE ID>
Win9x:
HKLM\Enum\<DEVICE ID>\<INSTANCE ID>.

This page is intentionally blank

C H A P T E R 4

Loading Driver

Loading by PnP Manager

The PCMCIA bus driver detects the inserted card , powers it on, and sends notification to PnP Manager. The PnP Manager requests **DEVICE ID** and **INSTANCE ID** of inserted card (see "Device ID Generation" and "Instance ID Generation"), and tries to locate the record for this card in the registry. If the record is present, the PnP manager loads the corresponding device driver, (if it is not already loaded) and sends an Add Device (Win9x: New Device) request. If the record is not present, the PnP manager starts Setup wizard (see "PnP Device Installation Issues")

WinNT OS

The PCMCIA driver detects the insertion of the card and asks the PnP Manager to send re-enumeration request to the socket device

IoInvalidateDeviceRelations(SocketPDO, BusRelations)

The PnP Manager sends an **IRP_MN_QUERY_DEVICE_RELATIONS** request to the socket PDO. The PCMCIA driver powers on the card, creates a PDO device, generates a DEVICE ID and INSTANCE ID, and returns the address of the PDO to the list of relations. The PnP Manager takes the name of the card driver from the registry, loads the driver and calls the AddDevice routine (see Win2K DKK "PnP and Power Management AddDevice Routine")

Note: If PCMCIA driver fails to generate DEVICE ID, PDO will be not created and request will be completed with status **STATUS_DEVICE_NOT_READY.** Card is invisible for PnP subsystem.

The reason of the problem is the incorrect algorithm of CIS parsing inside of the PCMCIA bus driver. The PCMCIA bus interface has a set of methods for reading the attribute memory. According to the PC Card Standard, if CIS is not presents in the attribute memory the interface will read the common memory. Unfortunately, during CIS parsing the PCMCIA bus driver doesn't validate the data, so it can interpret any kind of data as CIS. Depending on the data read from the common memory by the PCMCIA driver, it can cause:

- Return error from **IRP_MN_QUERY_DEVICE_RELATIONS** (for example, if common memory contains a lot of zero bytes, PCMCIA interprets them as multiple CISTPL_NULL and returns error)

- Generat invalid resource requirements list

Win9x OS

The PCMCIA driver detects the insertion of the card, powers it on, generates the DEVICE ID and INSTANCE ID, and then creates the device node to inform the PnP Manager that a new device was found

CM_Create_DevNode(&hNode, pszDeviceID, hSocket, 0)

The PnP Manager takes the name of the card driver from the registry, loads the driver and sends the message **PNP_NEW_DEVNODE** to the driver.

Static and Dynamic Loading of Driver

In some cases the driver must be loaded dynamically, and sometimes at boot of the system. For example, if driver supports both, PnP and Legacy cards, or installs some types of system hooks.

WinNT OS

Driver needs to be registered via Service Control Manager (see MSDN "OpenSCManager", "CreateService" and "ChangeServiceConfig"). Depending on dwStartType parameter passed to CreateService, Windows NT will start your driver at boot or system time, automatically. If start type is 'manual', ControlService function must be called directly to start the service.

Note: If driver registers AddDevice routine (it is required for PnP drivers), attempt to load driver with system; automatic and manual start will fail if

```
HKLM\SYSTEM\CurrentControlSet\Services\<Name of service>\Enum
      Count      REG_DWORD      0
```

'Count' value is zero. PnP manager increases this value when it detects hardware associated by "Found new hardware Wizard" with your driver.

The driver can use the following algorithm to avoid the described problem above:

1. DriverEntry: check Count value. If value is non-zero, register AddDevice routine. If Count is zero, register the re-initialization routine via **IoRegisterDriverReinitialization** call

2. When re-initialization routine is called, register AddDevice routine

Win9x OS

Dynamic load by ring 3 application

VxD driver can be loaded via ring 3 **CreateFile** call. In this operation mode the driver will receive a **SYS_DYNAMIC_DEVICE_INIT** message from the Virtual Machine Manager (VMM). After the last handle it will be closed by the **CloseHandle** call, and Win9x will unload VxD.

```
  // Load VxD
LPCTSTR lpszVxDName = “\\\\.\\C:\\WINDOWS\\AVxDName.VxD”;
HANDLE  hDevice =
        CreateFile(lpszVxdName,
                        0, 0, NULL, 0,
                        FILE_FLAG_DELETE_ON_CLOSE |
                        FILE_FLAG_OVERLAPPED, NULL);
…
// Unload VxD
CloseHandle(hDevice);
```

Static load

VMM32 examines the registry branch SYSTEM\CurrentControlSet\Services\VxD and enumerates all keys under this branch. ("Enumerate" here refers to the enumeration of registry keys, not to Plug and Play enumeration.) If it finds a value "StaticVxD=", it will load that static VxD and execute its real mode initialization portion. For example:

SYSTEM\CurrentControlSet\Services\VxD\AvxDName

Description=**Driver description string**

Manufacturer=**APSoft**

StaticVxD=**AvxDName**

Start=0

Refer to Win95 DDK “Loading Base Drivers Specified in the Registry” chapter for information on how to load VxD at system boot.

This page is intentionally blank

C H A P T E R 5

Driver Operation

WinNT OSs

There are three types of PnP drivers: bus, function, and filter.

Bus driver

A bus driver services a bus controller, adapter, or bridge. Microsoft provides a bus drivers for most common buses, such as PCI, PnpISA, SCSI, and USB. Other bus drivers can be provided by IHVs, or OEMs. Bus drivers are required drivers.

The primary responsibilities of a bus driver are to enumerate the devices on its bus, and respond to certain PnP and power management IRPs. During enumeration, a bus driver identifies the devices on its bus and creates device objects for them. The method a bus driver uses to identify connected devices depends on the particular bus.

A bus driver could be implemented as a driver/minidriver pair, the way a SCSI port/miniport pair drives a SCSI HBA (host bus adapter).

A bus driver manages the slots on its bus; a function driver manages a device that is attached to a bus. A bus driver is a function driver for its controller or adapter.

Function

A function driver is the main driver for a device and provides the operational interface for a device. A function driver typically handles read and writes to the device and manages device power policy.

A function driver creates an FDO and attaches it to the device stack. If a device is being used in raw mode it has no function driver, and all raw-mode I/O is handled by the underlying bus driver (and optional bus filter drivers).

The function driver for a device can be implemented as a driver/minidriver pair, such as a port/miniport pair or class/miniclass pair. In such driver pairs, one driver is linked to the second driver (which is a DLL).

Filter

An intermediate driver that intercepts and processes I/O requests bound for an underlying device. Such a driver calls IoAttachDevice at initialization to alias its own device objects to those of the underlying device driver(s) or to each intermediate driver layered above an underlying device driver.

Add device procedure

During **AddDevice** processing, driver should create new device object, which will receive all PnP subsystem IRPs for corresponding device. All these IRPs are described in Windows DDK (see “IRP_MN_xxx” topics), but the order of IRPs received by the device is not documented. For PCMCIA cards the order is as follows.

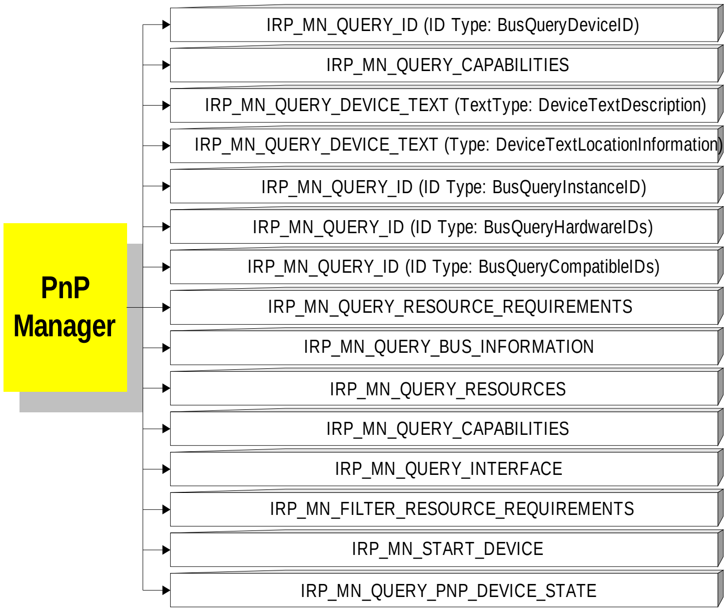
PnP Manager
IRP_MN_QUERY_ID (ID Type: BusQueryDeviceID)
IRP_MN_QUERY_CAPABILITIES
IRP_MN_QUERY_DEVICE_TEXT (TextType: DeviceTextDescription)
IRP_MN_QUERY_DEVICE_TEXT (Type: DeviceTextLocationInformation)
IRP_MN_QUERY_ID (ID Type: BusQueryInstanceID)
IRP_MN_QUERY_ID (ID Type: BusQueryHardwareIDs)
IRP_MN_QUERY_ID (ID Type: BusQueryCompatibleIDs)
IRP_MN_QUERY_RESOURCE_REQUIREMENTS
IRP_MN_QUERY_BUS_INFORMATION
IRP_MN_QUERY_RESOURCES
IRP_MN_QUERY_CAPABILITIES
IRP_MN_QUERY_INTERFACE
IRP_MN_FILTER_RESOURCE_REQUIREMENTS
IRP_MN_START_DEVICE
IRP_MN_QUERY_PNP_DEVICE_STATE

Figure 1: The list of IRP_MJ_PNP requests sent by PnP Manager to PDO device of PCMCIA card

IRPs sent to driver when device is added

IRP_MN_QUERY_ID (BusQueryDeviceID)

If a driver returns ID(s) in response to this IRP, it allocates a WCHAR structure from paged pool to contain the ID(s). The PnP Manager frees the structure when it is no longer needed.

The PnP Manager creates a sub-key in the CurrentControlSet\Enum key using the returned string.

IRP_MN_QUERY_CAPABILITIES

The PnP Manager sends this IRP to get the capabilities of a device, such as whether the device can be locked or ejected.

PnP Manager writes the value of LockSupported – SurpriseRemovalOK members to the Capabilities value of the \SYSTEM\CurrentControlSet\Enum\Pcmcia\<DEVICE ID>\<INSTANCE ID > key.

IRP_MN_QUERY_DEVICE_TEXT (DeviceTextDescription)

Bus drivers are strongly encouraged to return device descriptions for their child devices. This string is displayed in the "found new hardware" pop-up window if no INF match is found for the device.

Bus drivers are also encouraged to return location information for their child devices, but this information is optional.

If a bus driver returns information in response to this IRP, it allocates a NULL-terminated Unicode string from paged memory. The PnP Manager frees the string when it is no longer needed.

IRP_MN_QUERY_DEVICE_TEXT (DeviceTextLocationInformation)

PCMCIA driver returns the STATUS_NOT_SUPPORTED status when it received this request.

IRP_MN_QUERY_ID (BusQueryInstanceID)

PCMCIA returns the instance number of card (“1”, “2”, ...).

IRP_MN_QUERY_ID (BusQueryHardwareIDs)

PCMCIA returns the same string as from IRP_MN_QUERY_ID (BusQueryDeviceID).

IRP_MN_QUERY_ID (BusQueryCompatibleIDs)

PCMCIA can return one of the following strings: “*PNP0600”, “*PNPC200”, “*PNP0D00” or “”.

PnP Manager saves a returned string to the CompatibleIDs value of PCMCIA database.

IRP_MN_QUERY_RESOURCE_REQUIRENMENTS

The PnP Manager sends this IRP when a device is enumerated, prior to allocating resources to a device, and when a driver reports that its device's resource requirements have changed. Sometimes (e.g., for PCMCIA memory cards) bus driver returns NULL resource list for its PDO. So, functional driver should process this IRP.

IRP_MN_QUERY_BUS_INFORMATION

PCMCIA driver returns PCMCIABus value, when it received this request.

IRP_MN_QUERY_RESOURCES

A bus driver that handles this IRP sets Irp->IoStatus.Information to a pointer to a CM_RESOURCE_LIST that contains the requested information.

PCMCIA driver returns the STATUS_NOT_SUPPORTED status when it received this request.

IRP_MN_QUERY_INTERFACE

PnP Manager requests the following interfaces:

GUID_TRANSLATOR_INTERFACE_STANDARD

GUID_BUS_INTERFACE_STANDARD

PCMCIA driver does not support these interfaces.

IRP_MN_START_DEVICE

PCMCIA driver enables the PC card when it receives this request.

PnP Manager saves the address of device to the DeviceReference value in PCMCIA database.

IRP_MN_QUERY_PNP_DEVICE_STATE

If card is ready to work, driver returns READY value.

IRPs sent to driver when device is removed

If user uses “Unplug or Eject Hardware” applet, PnP Manager sends the:

a) **IRP_MN_QUERY_DEVICE_RELATIONS** (EjectionRelations) request to bus PDO device (Pcmcia0 device receives this request, because it is attached to device stack of PDO device)

b) **IRP_MN_QUERY_DEVICE_RELATIONS** with EjectionRelations and then with RemovalRelations type to the child PDO device

c) **IRP_MN_REMOVE_DEVICE** request to the child PDO device

PCMCIA does not destroy the PDO device under step c). It receives this request again when the card is physically removed from socket.

If card was not stopped using the “Unplug or Eject Hardware” applet and user removes it from socket, PCMCIA receives **IRP_MN_SURPRISE_REMOVAL** request (before **IRP_MN_REMOVE_DEVICE**).

Surprise removal of device

If you unsafely remove a card, Windows NT sends an
IRP_MN_SURPROSE_REMOVAL
and then
IRP_MN_REMOVE_DEVICE
request. **IRP_MN_REMOVE_DEVICE** request will not be sent if there is an open handle on PDO, and/or FDO devices are still attached. Other IRPs processing by driver

Refer to WinNT DDK “IRP_MJ_xx” chapter for information on how to process non-PnP requests.

Win9x OSs

VMM Messages and Control Procedure

Every virtual device needs a device control procedure. The VMM calls this procedure to send the virtual device system control messages. The system control messages direct the virtual device to carry out actions, such as initializing itself, and to notify the virtual device of changes to virtual machines, such as a virtual machine is being created. Most virtual devices define the device control procedure by using the Begin_Control_Dispatch, Control_Dispatch, and End_Control_Dispatch macros as in the following example:

```
Begin_Control_Dispatch VSAMPLED
    Control_Dispatch   Sys_Critical_Init,  VSAMPLED_Crit_Init
    Control_Dispatch   Device_Init, VSAMPLED_Device_Init
    Control_Dispatch   Sys_Critical_Exit, VSAMPLED_Crit_Exit
End_Control_Dispatch VSAMPLED
```

In this example, the macros create a device control procedure, named VSAMPLED_Control, and generate instructions that check for the messages Sys_Critical_Init, Device_Init, and Sys_Critical_Exit. When these messages are sent to the procedure, the procedure passes control to a corresponding procedure, such as VSAMPLE_Device_Init, to process the messages. The virtual device must define these message-processing procedures.

List of messages received by static VxD at boot of system:

SYS_CRITICAL_INIT

DEVICE_INIT

INIT_COMPLETE

SYS_VM_INIT

BEGIN_PM_APP

KERNEL32_INITIALIZED

List of messages received by VxD at restart of system:

DEVICE_REBOOT_NOTIFY

DEVICE_REBOOT_NOTIFY2

CRIT_REBOOT_NOTIFY

CRIT_REBOOT_NOTIFY2

List of messages received by dynamic VxD at startup:

SYS_DYNAMIC_DEVICE_INIT

List of messages received by dynamic VxD at unload:

SYS_DYNAMIC_DEVICE_EXIT

For more information refer to “Device Control procedure” and “Begin_Control_Dispatch” chapter in Windows 98 DDK help.

Connection to Configuration Manager

When a bus enumerator requests a new device node, the Configuration Manager locates the device ID of the device node in the system registry and loads the driver(s) using the registered device loader, if the necessary entries are present. After loading the device driver, the Configuration Manager sends a PnP_New_DevNode message to the driver's control procedure.

Note: If the new device node is the result of the "first insertion" event, the device node is setup during "appy-time." After successful installation of the device software, the Configuration Manager continues the startup sequence.

On receipt of the **PnP_New_DevNode** message, the device driver registers a "Config Handler" procedure with the configuration manager, using the **CONFIGMG_Register_Device_Driver** service. The Config Handler processes all subsequent CONFIG_ type messages from the Configuration Manager.

Configuration Manager Messages sent to driver when device is added

CONFIG_FILTER

Sent to direct a driver to process a new device or changes to the configuration. Enumerators should retrieve the filtered logical configuration for the indicated device node and modify the requirements as needed. For example, the PCMCIA driver removes unsupported IRQs and pre-allocates IO port and memory ranges.

Refer to “CONFIG_FILTER” chapter of Windows 98 DDK for more information

CONFIG_START

Sent when a configuration has been allocated for the device node. The driver can retrieve the allocated configuration and begin to use it.

CONFIG_ENUMERATE

Sent to direct a driver to enumerate its immediate children. This message is sent in response to the insertion or removal of a device. An enumerator should create a device node for each child by using the **CONFIGMG_Create_DevNode** service or remove children by using the **CONFIGMG_Remove_SubTree** service as appropriate.

Configuration Manager Messages sent to driver when device is removed

If user uses PCCARD control to stop device, then CM manager will send **CONFIG_TEST** message to request possibility of device stopping. On physical card removal CM manager will send **CONFIG_PREREMOVE**, **CONFIG_PREREMOVE2** and **CONFIG_REMOVE** messages to PnP dispatcher. The first two messages notify driver, that it should stop using device’s configuration. The last message notifies driver, that device was removed.

PnP dispatcher will receive **CONFIG_PRESHUTDOWN** and **CONFIG_SHUTDOWN** when Windows will shutdown.

Other Configuration Manager messages processed by driver

CONFIG_APM

Sent to notify a driver of a power management event. The driver should determine the type of event and take appropriate action.

CONFIG_SETUP

Sent to notify the driver that the device node has been setup. The driver should load additional drivers if possible. For example, the driver could load appropriate drivers from device ROM, such as for ISA_RTR, PCI and PCMCIA device.

CONFIG_READY

Sent to notify the driver that the associated device node has been set up.

C H A P T E R 6

Specific Problems under WinNT OS

Resource Allocation

There are possible resource problems due to the Windows 2000/XP PnP architecture. The main problem is that some BIOS can’t initialize PCI-2-PCI bridges properly. If the CardBus adapter is locates behind the PCI-2-PCI Bridge or the inserted CardBus card is a bridge, then resources, assigned to the CardBus adapter can be incomplete. For example, resources, assigned to the CardBus adapter can only cover 256 bytes of I/O space. But if your CardBus card is a PCI-2-PCI Bridge, then you will need at least 4KB of I/O space. Since Windows 2000/XP PnP subsystem can’t restart bridges to update resources, you will not be able to configure your card correctly. On the other hand, your CardBus adapter can be located behind the PCI-2-PCI Bridge (as on some modern PC). In such a situation, resources of your card should be allocated inside of resources of all parents bridges. It can cause additional problems of resource allocation.

The other problem is the support of legacy resources or VGA resources on bridges. The ISA enable bit and VGA enable bit are present inside of the PCI configuration space of the bridge device. s. These bits are responsible for passing legacy resources through the bridges.

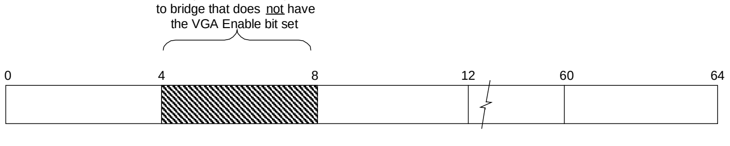

Figure 2: Allocation for a bridge without VGA child device.

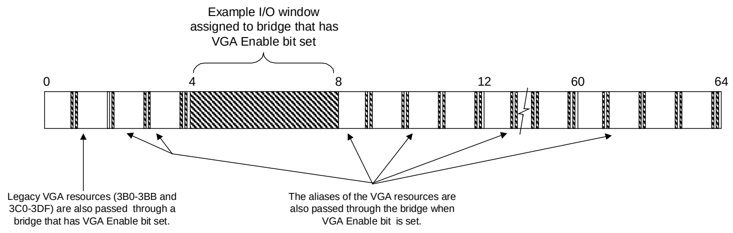
Example I/O window
assigned to bridge that has
VGA Enable bit set
0
4
8
12
60
64
Legacy VGA resources (3B0-3BB and 3C0-3DF) are also passed through a bridge that has VGA Enable bit set.
The aliases of the VGA resources are also passed through the bridge when VGA Enable bit is set.

Figure 3: A bridge that has the VGA enable bit set

When the VGA enable bit is set, then the PCI-2-PCI Bridge receives all requests to the legacy ISA resources (3B0-3BB for VGA monochrome and 3C0-3DF for VGA color). The enable bit should also force the bridge to pass all 16-bit aliases of the VGA resources to avoid resource conflicts. Figures 2 and 3 demonstrate the whole I/O space, splitt on the 4KB ranges (as required for bridge devices). Every 4KB range has several aliases for VGA legacy resources. Such condition greatly decreases the amount of available I/O space.

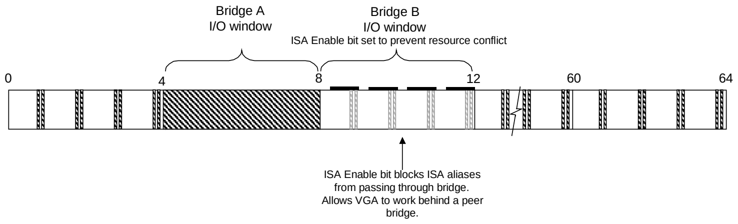
Bridge A
I/O window
Bridge B
I/O window
ISA Enable bit set to prevent resource conflict
0
4
8
12
60
64
ISA Enable bit blocks ISA aliases from passing through bridge. Allows VGA to work behind a peer bridge.

Figure 4: A bridge with the ISA enable bit set

The ISA Enable bit prevents conflicts between 16-bit aliases of VGA resources (and other legacy resources) and PCI-to-PCI bridge windows. When a bridge has the ISA Enable bit set, 16-bit aliases of 100h – 3FF are blocked from passing through the bridge.

Figure 4 shows the effect of setting the ISA Enable bit of the bridge (Bridge A). This blocks ISA aliases from passing through the bridge and prevents conflicts between bridge A and Bridge B (although it still consumes approximately 75% of the I/O range allocated to the bridge).

Drive Letters Mounting / Unmounting

Later, in Windows NT OS, you can create a drive letter for your memory device, by using a symbolic link name for your device name. Under Windows 2000/XP you have the possibility to assign a drive letter, using the standard component – Mount Manager. MM is responsible for managing volume names. MM API declared in Windows 2000 DDK's headers: *mountmgr.h* and *mountdev.h*. To use it from your driver, you should register MM interface via **IoRegisterDeviceInterface** call with **MOUNTDEV_MOUNTED_DEVICE_GUID**. This component will send 3 requests to the driver:

- **IOCTL_MOUNTDEV_QUERY_DEVICE_NAME (Optional)** – driver should return its device name (*see **MOUNTDEV_NAME** structure*).
- **IOCTL_MOUNTDEV_QUERY_UNIQUE_ID (Required)** – driver should return symbolic link name, received in **IoRegisterDeviceInterface** call (*see **MOUNTDEV_UNIQUE_ID** structure*).
- **IOCTL_MOUNTDEV_QUERY_SUGGESTED_LINK_NAME (Optional)** – driver should suggested link name (*see **MOUNTDEV_SUGGESTED_LINK_NAME** structure).*

MM internally checks the possibility of suggested assigned link name to device and will use this link, if it's possible and registry database (**HKLM\SYSTEM\MountedDevices**) doesn't have a record for the device. Otherwise, MM will use available drive letter or drive letter, which presents in database.

How WinNT OS Sees CardBus Adapters

If the CardBus adapter has more then one socket, Windows NT works with each socket as if they were different adapters.

CardBus cards support

If your adapter is a CardBus adapter, then it supports CardBus cards. Since the CardBus interface is physically and logically compatible with the PCI interface, the PCMCIA driver simply creates a new PCI bus, using the PCI bus driver interfaces, and redirects all the card's requests to the PCI driver. For CardBus cards the PCMCIA driver doesn't try to locate CIS, since the PCI driver simply recognizes the card, using the PCI configuration space of the card.

This page is intentionally blank

C H A P T E R 7

Specific Procedures Illustrated in Samples

WinNT OS

WinNT sample builds functional (FUNCDRV.SYS) and filter (FLTDRV.SYS) drivers for a memory card and application, which are able to communicate with a functional driver by sending I/O control requests.

Filter Driver

This driver creates and attaches a filter device. It forwards all IRP requests to the lower device, except a Filter Resource requirements request. The handler of filter resource requirements request forwards IRP to lower device (functional driver), then modifies Alignment and Length of memory resource descriptor.

Function Driver

The driver creates a device object and attaches it to the PDO device. It forwards all IRP requests to the lower device, except a Filter Resource requirements request.

- Handler of filter resource requirements request forwards IRP to lower device (PCMCIA driver), then replaces Resource Requirements list to request 8KB memory window.

- Handler of start device request saves address of assigned 8KB window in a member variable.

It also contains simple implementation of IRP_MJ_READ and IRP_MJ_WRITE requests.

Application

The application displays a dialog box that asks the user to select one of the following actions: Query ID (synchronous), Query ID (asynchronous) or Get Version. When an action is selected, the application sends device I/O control to the function driver and displays returned data in the edit box control.

Bulk

This client application monitors all card insertion/removal events from **“CardWare 7.0 Memory Cards"** driver. If Memory card is inserted (SRAM, FLASH, ATA), first 512 bytes of first memory region will be printed in order to demonstrate usage of the driver memory services.

Source Description

Win2K\Driver directory contains implementation of base classes for PnP drivers.

PnPDrv.h	Definition of CPnpDriver class.
PnPDrv.cpp	Implementation of EntryPoint, DriverUnload and CPnPDriver class
PnPDev.h	Definition of CPnPDevice class
PnPDev.cpp	Implementation of CPnPDevice class.
PnPMn.cpp	Default PnP request handlers
Utils.cpp	Set of functions for operating with Unicode string and sending PnP requests to a driver
Deb_Res.cpp	Functions for tracing resource lists

Win2K\Driver\Filter directory contains source of FLTDRV.SYS driver.

DrvClass.h	Definition of CFltDriver class derivative from CPnpDriver class.
DrvClass.cpp	Implementation of CFltDriver class. Overloaded method of CPnpDriver class –OnAddDevice - creates object of CFltDevice class
DevClass.h	Definition of CFltDevice class derivative from CPnpDevice class.
DevClass.cpp	Implementation of CFltDevice class. OnFilterResReq method modifies memory descriptor of resource requirements list.

Win2K\Driver\Function directory contains source of FUNCDRV.SYS driver.

DrvClass.h	Definition of CFuncDriver class derivative from CPnpDriver class.
DrvClass.cpp	Implementation of CFuncDevice class. Overloaded method of CPnpDriver class – OnAddDevice - creates object of CFltDevice class
DevClass.h	Definition of CFuncDevice class derivative from CPnpDevice class.
DevClass.cpp	Implementation of CFuncDevice class. OnFilterResReq method replaces resource requirements list.
DevIoctl.cpp	Implementation of I/O control handlers

Win9x OS

This is a sample of a generic PCMCIA card driver with Card Services support and communication application.

The Configuration Manager loads the driver when it detects an insertion of a card described in the INF file. The driver registers its self as a client of the card Services and waits for PnP_New_DevNode notification. After notification if received, the driver registers the configuration handler. CONFIG_START handler starts the adapter.

Source Description

Startup.asm	VxD Control procedure
Main.cpp, Main.h, Vxdmap.h, Vxdmap.cpp	Processing of VMM notifications
PnPDisp.cpp	Processing of Configuration Manager messages
Filer.h, Filer.asm, Filer.inc	Access to Disk files
Registry.cpp	Access to registry
Cs.h, Cs.cpp, Pcmcia.cpp	Access to card Services API
Heap.cpp	Heap management
Error.cpp	Error output
Win32api.cpp	Win32API for applications
Wrapper.asm	Real mode wrappers

C H A P T E R 8

CardWare 7.0 Card Services API

Card Services coordinates access to PC Cards, sockets and system resources among multiple clients. These clients may be resident or transient device drivers, system utilities, or application programs.

Card Services preserves for its clients an abstract, socket-hardware-implementation independent view of a card and its resources. Card Services presents the same tuple organizational and resource allocation view to all of its clients whether the card is a 16-bit PC Card or a CardBus PC Card.

CardWare 7.0 provides Card Services level 2.10 under Windows 98/ME and Card Services level 5.02 under Windows 2000/XP.

See below table with list of supported functions under Windows 98/ME and Windows 200/XP.

Function		Win98/ME	Win2K/XP
#	Name	PCMCIA 2.10	PCMCIA 5.02
00	CS_CLOSEMEM	+	+
01	CS_COPYMEM	+	+
02	CS_DEREGCLIENT	+	+
03	CS_GETCLIENTINFO	+	+
04	CS_GETCONFIGINFO	+	+
05	CS_GETFIRSTPART	+	+
06	CS_GETFIRSTREGION	+	+
07	CS_GETFIRSTTUPLE	+	+
08	CS_GETNEXTPART	+	+
09	CS_GETNEXTREGION	+	+
0A	CS_GETNEXTTUPLE	+	+
0B	CS_GETCSINFO	+	+
0C	CS_GETSTATUS	+	+
0D	CS_GETTUPLEDATA	+	+
0E	CS_GETFIRSTCLIENT	+	+
0F	CS_REGERASEQ	+	+
10	CS_REGCLIENT	+	+
11	CS_RESETCARD	+	+
12	CS_MAPLOGSKT	+	+
13	CS_MAPLOGWND	+	+

14	CS_MAPMEMPAGE	+	-
15	CS_MAPPHYSKT	+	+
16	CS_MAPPHYWND	+	+
17	CS_MODIFYWND	+	-
18	CS_OPENMEM	+	+
19	CS_READMEM	+	+
1A	CS_REGMTD	+ (Only for VxD)	+
1B	CS_RELEASEIO	+ (Only for VxD)	-
1C	CS_RELEASEIRQ	+ (Only for VxD)	-
1D	CS_RELEASEMEM	+ (Only for VxD)	-
1E	CS_RELEASECONFIG	+ (Only for VxD)	-
1F	CS_REQUESTIO	+ (Only for VxD)	-
20	CS_REQUESTIRQ	+ (Only for VxD)	-
21	CS_REQUESTMEM	+ (Only for VxD)	-
22	CS_REQUESTSKTMSK	+	+
23	CS_RETSSENTRY	+	+
24	CS_WRITEMEM	+	+
25	CS_DEREGERASEQ	+	+
26	CS_CHECKERASEQ	+	+
27	CS_MODIFYCONFIG	+ (Only for VxD)	-
28	CS_REGTIMER	+	+
29	CS_SETREGION	+	+
2A	CS_GETNEXTCLIENT	+	+
2B	CS_VALIDATECIS	+	+
2C	CS_REQUESTEXCL	+	+
2D	CS_RELEASEEXCL	+	+
2E	CS_GETEVENTMSK	+	+
2F	CS_RELEASESKTMSK	+	+
30	CS_REQUESTCONFIG	+ (Only for VxD)	-
31	CS_SETEVENTMSK	+	+
32	CS_ADDSS	+ (Only for VxD)	+
33	CS_REPLACESS	+ (Only for VxD)	+
34	CS_VENDORSPECIFIC	+	+
35	CS_ADJUSTRESINFO	-	-
36	CS_ACCESSCONFIGREG	+	+
37	CS_GETFIRSTWINDOW	+	+
38	CS_GETNEXTWINDOW	+	+
39	CS_GETMEMPAGE	-	+
3A	CS_REQUESTDMA	-	-
3B	CS_RELEASEDMA	-	-
3C	CS_CONFIGURE_FUNCTION	-	-

3D	CS_INQUIRE_CONFIGURATION	-	-

See below table with list of supported client callback events

Event		Win98/ME	Win2K/XP
#	Name		
01	BATTERY_DEAD	+	-
02	BATTERY_LOW	+	-
03	CARD_LOCK	+	-
04	CARD_READY	+	-
05	CARD_REMOVAL	+	+
06	CARD_UNLOCK	+	-
07	EJECTION_COMPLETE	-	-
08	EJECTION_REQUEST	-	-
09	INSERTION_COMPLETE	-	-
0A	INSERTION_REQUEST	-	-
0B	PM_RESUME	-	-
0C	PM_SUSPEND	-	-
0D	EXCLUSIVE_COMPLETE	+	+
0E	EXCLUSIVE_REQUEST	+	+
0F	RESET_PHYSICAL	+	+
10	RESET_REQUEST	+	+
11	CARD_RESET	+	+
12	CS_EV_MTD_REQUEST	+	+
14	CLIENT_INFO	+	+
15	TIMER_EXPIRED	+	+
16	SS_UPDATED	+	+
17	WRITE_PROTECT	+	-
18	REQUEST_ATTENTION	-	-
40	CARD_INSERTION	+	+
80	RESET_COMPLETE	+	-
81	ERASE_COMPLETE	+	+
82	REGISTRATION_COMPLETE	+	+
F0	POST_COMPLETE	-	-
F1	PWR_CONSUMPTION_CHANGE	-	-
F2	CS_EV_MTD_REQUEST_SEC	+	+

Bulk memory services

Bulk Memory Services provide services that can be used by clients such as file system utilities or XIP install utilities to avoid dealing with all the details of the various memory technologies that can be present on PC Cards. These services support a simple Open/CloseMemory and Read/Write/CopyMemory model of memory access. This model is similar to open, close, read, and write access to files in most operating systems.

Card Services determines PC Card memory regions during card insertion processing. Clients may determine areas in PC Card memory they wish to access by parsing the CIS or using the Card Services' services. GetFirst/NextTuple, GetFirst/NextPartition, or GetFirst/NextRegion. Once a client determines the area of the PC Card they wish to access, they use the OpenMemory request and specify the absolute offset on the PC Card where the area begins.

OpenMemory returns a memory handle that is used for all subsequent read, write, copy and erase operations. These operations specify the location to be accessed relative to the start of the opened memory area. This allows clients to move data to and from PC Card memory as desired without concern as to where on the PC Card this particular memory area lies. A client performs a CloseMemory request to inform Card Services that it will no longer be accessing a memory area.

Performing an erase operation differs from the read, write and copy services. A client that needs to erase memory must register an erase queue with the RegisterEraseQueue request. The client then fills an erase queue entry identifying the socket and region of memory to erase. Next, the CheckEraseQueue request is used to notify Card Services that one or more erase requests have been made in the erase queue. The actual erase operation is performed asynchronously. When the erase operation completes, the client is notified through the callback entry point provided in the erase queue header. In comparison, the read, write, and copy services return only after the requested action has been completed.

A client must use DeregisterEraseQueue to request Card Services to relinquish control of an erase queue. This must be invoked before a client is removed from memory. DeregisterEraseQueue can only be used when there are no queued erase requests in the erase queue.

Since erase operations return before the erase is complete, the client may be able to access other services while waiting for the erase completion. The physical construction of some cards (or memory components) may prevent some services until the erase operation in progress has been completed. In this event, the other requested operation is blocked (delayed) within Card Services until the erase is completed. Card Services does not notify the requester of erase completion until the blocked request is complete.

Description of samples using Bulk Memory services can be found in CWSDK document.

Native memory cards support API of CardWare 7.0

In addition to Card Services interface CardWare 7.0 for Windows 2000/XP provides an API (native) allowing to software developer write his own applications exploring memory cards.

Driver programming interface

Include files description:

..\Src\Inc\CWMEM2K.h	All IOCTL codes and corresponding structures

Driver's control device, which actually receives all IOCTLs, is named "CWMEM2K".

Below you can see an example of calling `IOCTL_FLASH2K_GETVERSION:`

```
****************************************************

HANDLE h;
BOOL   fSuccess;

h = CreateFile("\\\\.\\CWMEM2K", 0,
               FILE_SHARE_READ | FILE_SHARE_WRITE,
               NULL, OPEN_EXISTING, 0, NULL);
```

Check driver version

```
fSuccess = DeviceIoControl(h, IOCTL_FLASH2K_GETVERSION, NULL,
0,
                     &dwVersion, sizeof(dwVersion),
                     NULL);

if (!fSuccess)
{
   // Error handling
}
else
{
   // Analyze version info
}

***************************************************
```

Informational functions

IOCTL_FLASH2K_GETVERSION

Operation:	Returns driver version
Input:	Not used
Output:	DWORD
Comment:	Returns four bytes of driver version. First byte – major version, second byte – minor version. Third and fourth bytes – build number

IOCTL_FLASH2K_GET_SOCKET_MASK

Operation:	Retrieves information about sockets/cards in the sockets
Input:	Not used
Output:	FLASH2K_SOCKET_MASK structure
Comment:	Returns mask of sockets in dwMask member and mask of sockets with inserted flash cards in dwFlashMask member. Client application of CWMEM2K can send this request periodically to detect card insertion/removal.

IOCTL_FLASH2K_GET_SOCKET_INFO

Operation:	Retrieves socket information, like CIS information, type of media in the socket, etc
Input:	FLASH2K_IOCTL structure. Following fields must be filled out before send the IOCTL: bDrive or wSocket and fSkt. If fSkt is set to BOOL_TRUE, CWMEM2K wSocket value.
Output:	SKT_INFO structure
Comment:	Returns information about type of inserted card and card CIS

IOCTL_FLASH2K_IDENTIFY_CARD

Operation:	Returns card identification information
Input:	FLASH2K_IOCTL structure. Following fields must be filled out before send the IOCTL: bDrive or wSocket and fSkt. If fSkt is set to BOOL_TRUE, CWMEM2K wSocket value.
Output:	Buffer with card identification information, depended on card type: FLASH2K_IDENTIFY_INFO (Flash card) or SRAM_IDENTIFY_INFO (SRAM card), or ATA_IDENTIFY_INFO (ATA card).
Comment:	Returning information includes power information and description of partitions. Type of returning structure depends from type of inserted card.

IOCTL_FLASH2K_GET_MEDIA_INFO

Operation:	Returns general card information
Input:	FLASH2K_MEDIAINFO structure. Following fields must be filled out before send the IOCTL: bDrive or wSocket and fSkt
Output:	FLASH2K_MEDIAINFO structure
Comment:	Returns information about media size, status, JEDEC ID

IOCTL_FLASH2K_GET_STATUS

Operation:	Returns card status
Input:	FLASH2K_CARDSTATUS structure. Following fields must be filled out before send the IOCTL: bDrive or wSocket and fSkt
Output:	FLASH2K_CARDSTATUS structure. Only one status value is now supported - CS_WRITE_PROTECT
Comment:	Returns status: 0 (READY) or CS_WRITE_PROTECT

IOCTL_FLASH2K_DEV_INSTANCE

Operation:	Retrieves whole device instance name for the card inserted in the specified socket
Input:	FLASH2K_IOCTL structure. Following fields must be filled out before send the IOCTL: bDrive or wSocket and fSkt
Output:	WCHAR array
Comment:	Returns Unicode string with whole device instance name that can be used to retrieve device node via CM_Locate_DevNode(). Device node normally is used in all CM_xxx routines

CIS access functions

CWMEM2K contains code, which allows it to parse CIS on memory cards to locate some information (like device description, configuration information, etc.). The same code can be used to enumerate tuples in CIS.

Please, remember, that CWMEM2K doesn’t validate CIS before parsing it. So, tuple data can be invalid.

To give possibility of CIS enumerating, CWMEM2K provides 3 IOCTLs.

IOCTL_FLASH2K_GET_FIRST_TUPLE

Operation:	IOCTL receives GFNTARGS structure and returns the same structure
Input:	GFNTARGS structure, *HlogSkt* field - Logical socket number (0, 1, etc.)
Output:	GFNTARGS structure
Comment:	Returns *TupleCode* field - Code of the first tuple in CIS. *TupleLink* field - Link value of the first tuple in CIS.

IOCTL_FLASH2K_GET_NEXT_TUPLE

Operation:	IOCTL receives GFNTARGS structure and returns the same structure
Input:	GFNTARGS structure, *HlogSkt* field - Logical socket number (0, 1, etc.). *CIS* field - Current CIS pointer. You should use the same structure for all GET_FIRST / GET_NEXT / GET_DATA iterations to have correct CIS pointer. NEVER CHANGE THIS VALUE MANUALLY!
Output:	GFNTARGS structure
Comment:	Returns *TupleCode* field - Code of the first tuple in CIS. *TupleLink* field – Link value of the first tuple in CIS.

IOCTL_FLASH2K_GET_TUPLE_DATA

Operation:	IOCTL receives GTDARGS structure and returns the same structure
Input:	GFNTARGS structure, *HlogSkt* field - Logical socket number (0, 1, etc.). *DataMax* - Size of *TupleData* field *CIS* field - Current CIS pointer (see notes for previous IOCTL)
Output:	GFNTARGS structure
Comment:	Returns *DataLen* field - The whole size of tuple body. *TupleData* field - Tuple data (should be correctly allocated and *DataMax* field should be initialized)

Raw card access

IOCTL_FLASH2K_READ_MEMORY

Operation:	Reads specified amount of bytes from specified offset on the card
Input:	FLASH2K_READ structure and buffer for data. Following fields must be filled out before send the IOCTL: bDrive or wSocket and fSkt, dwOffset and dwLength
Output:	Copy of card's memory, read from specified offset on the card.
Comment:	Size of buffer must be same or greater than specified in the dwLength field of FLASH2K_READ.

IOCTL_FLASH2K_WRITE_MEMORY

Operation:	Writes specified amount of bytes to the specified offset on the card
Input:	FLASH2K_WRITE structure, buffer with data to write. Following fields must be filled out before send the IOCTL: bDrive or wSocket and fSkt, dwOffset and dwLength
Output:	None
Comment:	Size of buffer must be the same or greater than specified in dwLength field of FLASH2K_WRITE.

IOCTL_FLASH2K_FTL_SUPPORT

Operation:	Switches on/off FTL support for raw read/write operations
Input:	BOOL. TRUE – to turn FTL support ON, FALSE – to turn it OFF.
Output:	Not used.
Comment:	Supported only for FLASH cards.

Card Erase support

CWMEM2K implements fast erasing scheme, which gives possibility to greatly decrease time of FLASH card erasing. The main idea is write erase command to the several erase blocks simultaneously and then check erase status on all blocks in the cycle. But it was found, that FLASH card can remove RDY signal for short time immediately after writing erase command to the block and writing erase command to the other block cause to increase interval till RDY signal assertion. It cause PCMCIA.SYS driver to freeze PC on several seconds.

By this reason, CWMEM2K driver waits for RDY signal after writing erase command to the first block and before writing to the next block. RDY signal waiting works, using either internal PCMCIA.SYS call, or manually via sockets registers. In the first case we need to know address of such call and in the second case we need to know base address of registers. All these information can be retrieved from device extension of PDO (card) or FDO (socket).

Card erase algorithm:

1. Send IOCTL_FLASH2K_TEST_FAST_ERASE to check if fast erase algorithm is supported
2. Send IOCTL_FLASH2K_GET_MEDIA_INFO to retrieve size of card and erase block.
3. If no, send IOCTL_FLASH2K_ERASE_BLOCK request for each block (0, 1, …, dwCardSize / dwEraseBlockSize).
4. If yes, send IOCTL_FLASH2K_ERASE_CARD request. Send IOCTL_FLASH2K_ERASE_STATUS request in a loop. Break loop when status of each block (0, 1, …, dwCardSize / wEraseBlockSize) becomes equal to ERBLK_COMPLETE

IOCTL_FLASH2K_TEST_FAST_ERASE

Operation:	Test if fast erase algorithm is supported.
Input:	Not used.
Output:	BOOL. TRUE means that ‘known’ version of PCMCIA.SYS is used and fast erase algorithm is supported.
Comment:	Supported only for FLASH cards.

IOCTL_FLASH2K_ERASE_CARD

Operation: Erases whole card using fast erase algorithm.

Input: FLASH2K_IOCTL structure. Following fields must be filled out before send the IOCTL: fSkt, bDrive or wSocket

Output: BOOL. TRUE indicates that operation succeeds

Comment: Supported only for FLASH cards.

IOCTL_FLASH2K_ERASE_STATUS

Operation: Returns erase status for each erase block on the card.

Input: FLASH2K_IOCTL structure. Following fields must be filled out before send the IOCTL: fSkt, bDrive or wSocket.

Output: Array of bytes. Size of array (in bytes) must be equal number of erase blocks on the card. Each byte of the buffer contains erase status of corresponding erase block. The following erase status codes are supported: ERBLK_NOT_PROCESSED, ERBLK_IN_PROGRESS, ERBLK_FAILED, ERBLK_SUCCESS, ERBLK_COMPLETE.

Comment: Supported only for FLASH cards.

IOCTL_FLASH2K_ERASE_INT

Operation: Interrupts card erase.

Input: FLASH2K_IOCTL structure. Following fields must be filled out before send the IOCTL: fSkt, bDrive or wSocket.

Output: Not used.

Comment: Supported only for FLASH cards.

IOCTL_FLASH2K_ERASE_BLOCK

Operation:	Erases one block on FLASH card.
Input:	FLASH2K_ERASEBLOCK structure. Following fields must be filled out before send the IOCTL: fSkt, bDrive or wSocket, dwOffset, dwSize.
Output:	FLASH2K_ERASEBLOCK structure.
Comment:	Supported only for FLASH cards.

IOCTL_FLASH2K_CHECK_ERASE_BLOCK

Operation:	Checks if block is fully erased
Input:	FLASH2K_CHECKERASE structure. Following fields must be filled out before send the IOCTL: fSkt, bDrive or wSocket, dwMask, dwOffset. dwMask contains 0x0 or 0xFFFFFFFF depending on card type – on some FLASH cards erased blocks contains 0, on some – FF.
Output:	FLASH2K_CHECKERASE structure.
Comment:	Supported only for FLASH cards.

Utility functions

IOCTL_FLASH2K_RESET_SOCKET

Operation:	Resets socket.
Input:	FLASH2K_IOCTL structure. Following fields must be filled out before send the IOCTL: fSkt, bDrive or wSocket.
Output:	BOOL. TRUE indicates that operation succeeds.
Comment:	Initiates soft reset of socket