Driver Development

Version 2.1

GEOS Software Development Kit Library

Version 2.1

Driver Development

Geoworks
Alameda, CA

Driver Development

Driver Development

Driver Basics

1

DDK

There are three kinds of geode: applications, libraries, and drivers. Most programmers will only write applications. A few will write libraries, either for their own use or for other programmers'. A very few will write device drivers.

GEOS fully supports writing device drivers in assembly language. (It is possible to write some drivers in C, but this is not recommended, due to speed requirements.) Often, writing a driver for a new device is much like writing a driver for an existing, older device of the same kind; e.g. writing a driver for a new bus mouse is much like writing a driver for any other bus mouse. For this reason, many GEOS device drivers share a lot of code; e.g. most of the mouse drivers use a standard suite of routines, perhaps modified slightly for the particular mouse.

The SDK contains examples of mouse, power, pcmcia, and sound drivers; by examining them, you should be able to see how a driver is put together, and how to rewrite one of those drivers for a new device. These examples are located in the \OMNIGO\DRIVER\DDK directory.

1.1 Driver Basics

One of the advantages of the GEOS operating system is that it insulates application developers from many of the low-level hardware chores. GEOS does this by breaking up geodes into several types.

Most programmers write *applications*; these are the most visible geodes. Users interact directly with applications; as a general rule, they only interact indirectly with non-application geodes.

Applications use *libraries*. All applications use the GEOS kernel, which functions as a kind of "system library". Most applications use many other libraries as well. Libraries can have many functions. For example, object libraries define classes for applications to use, and many code libraries provide suites of pre-written routines that applications can use as needed. There is another role libraries play, though: they serve as an intermediary between *applications* and *drivers*.

Drivers provide the nuts-and-bolts interface between GEOS and the computer's devices and peripherals. Drivers take care of such tasks as writing data to the screen, sending information to the printer, and handling input from the mouse and keyboard. Drivers do not, as a rule, interact directly with applications or with the user. Instead, the kernel and other libraries act as intermediaries between the drivers (and their associated hardware) and the applications.

1.1

Each kind of driver (mouse, printer, power management, etc.) has certain functions it is expected to fulfill. From the point of view of a library using a printer driver, for example, all printer drivers should look and act more or less the same. Thus, a driver will have close constraints on its interface with other parts of GEOS.

1.1.1 Driver Behavior

In some ways, drivers behave differently from other geodes. One major difference is that for some drivers, running speed is a much higher priority than it is for most applications or libraries. A mouse driver, for example, has to handle mouse movements as quickly as possible, to keep from slowing down the rest of the system every time the user moves the mouse. For this reason, drivers tend to be written differently than other geodes. Most geodes are concerned with making their running size as small as possible; a driver is more likely to tolerate a larger size to get faster response. (This depends, of course, on the driver. A print driver, for example, doesn't need to be nearly as efficient as a video driver.)

Drivers may also disable interrupts to perfrom their functions. They are the only geodes permitted to do this. However, drivers should do this only when absolutely necessary. Very few GEOS system routines may be called with interrupts disabled. In practice, you should re-enable interrupts before making any system calls.

Drivers tend to use fixed memory more often than other geodes do. A driver's strategy routine (described below) and interrupt handlers must all be in fixed memory; any other timing-critical routines may also be in fixed resources.

Driver Development

1.1.2 Driver Structure

A driver usually provides two communication structures. One is the interface it provides to the GEOS system, either through a library or a GEOS application; the other is its contact with whatever device it drives.

The driver's interface to the system is called the *strategy routine*. Whenever the system or a library needs to interact with a driver, it calls the strategy routine; it passes in a code number saying what it needs to have the driver do. This code is used by a jump table to call an appropriate routine in the driver. The strategy routine must be in a fixed code resource.

A driver's interface with its device will depend on what kind of device it's driving. Most often, a driver will receive and handle interrupts from its device. It does this by registering an interrupt handler with the GEOS kernel. The interrupt handler must also be in a fixed code resource.

1.1.3 Extended Drivers

Some drivers can handle a number of similar, but distinct, devices. These drivers are known as *extended drivers*. An extended driver must provide certain extra information about itself to the system. Furthermore, it must provide certain extra functionality to the kernel.

When an extended driver is loaded, it is told which of its various devices it is intended to drive.

1.2 Defining a Basic Driver

Every driver, of whatever type, has a few components in common. The driver stores information about itself in a certain, rigidly defined way; that way, the system can get information about the driver in a direct manner.

The driver must have a strategy routine that performs certain specific functions. Some of these functions are similar for all drivers of whatever type; these are listed in this section. Others are specific for a type of driver; these will be listed in the chapter pertaining to that type of driver.

1.2.1 The Driver's .GP File

A driver is a geode. As such, it will be compiled as any other geode, and contains a geode parameters file. The geode parameters file looks somewhat different than an application **.gp** file, however. A driver's geode parameters file should exhibit the following characteristics:

- *name* should end with the **.drvr** suffix. This name must be unique across all drivers, applications, and libraries.
- *type* should be declared as *driver*. The driver may also be declared *system* and *single*.

  Marking a geode as *driver* results in the setting of GA_DRIVER in the geode's **GeodeAttrs** field. This instructs the system to call the driver's DR_INIT and DR_EXIT functions as appropriate.

  system indicates that the kernel relies upon the driver; it should remain loaded as long as possible if the system is shutting down. (This has ramifications for your DR_EXIT routine, discussed in "The DriverFunction Type" on page 20.)

  Most drivers will also want to be marked *single*, though if a library loads a driver using **GeodeUseDriver()**, the driver will behave as single-launchable regardless.
- *library* should declare any libraries that this driver needs to load. For example, a PCMCIA driver needs to include the *pcmcia* library.
- The resource that contains the driver's strategy routine, if it is not within **dgroup**, needs to be declared as *fixed*, *code*, *shared*, and *read-only*. Most other resources should be movable, if possible.

For information about any specific kind of driver, see the appropriate chapter, if available, and check the device's driver include file in \OMNIGO\INCLUDE\INTERNAL. All device include file names end with **dr.def**; for example, all mouse drivers must include the file **mousedr.def**.

1.2.2 Information about the Driver

`DriverTable, DriverInfoStruct, DriverAttrs, DriverType,`

As mentioned before, a driver contains one common routine entry point: its strategy routine. All functions executed by the driver are accessed through this routine. The driver does this by interpreting the function code passed to this strategy routine and executing another routine (through use of a jump table) upon determining what the driver should do. The address of this common strategy routine is contained within a **DriverTable** structure.

The **DriverTable** must reside in fixed memory. In most cases, this is accomplished by placing the table within **dgroup**. However, on XIP systems where **dgroup** needs to be marked discardable, the table should reside in a read-only, fixed, code resource. (In the unlikely case that your **dgroup** consists almost entirely of just a driver table, it is fine to leave the table in **dgroup** and leave it non-discardable.)

The name **DriverTable** is significant. The linker searches for this table, and locates the strategy routine in this manner. The first item within your **DriverTable** must be a **DriverInfoStruct** structure. This structure contains certain basic information about the driver, including the location of the strategy routine.

The **DriverInfoStruct** has the following definition:

```
DriverInfoStruct  struct
      DIS_strategy            fptr.far
      DIS_driverAttributes    DriverAttrs
      DIS_driverType          DriverType
DriverInfoStruct                ends
```

DIS_strategy This is the address of the strategy routine. The strategy routine must be in a fixed code resource.

DIS_driverAttributes
This field contains a **DriverAttrs** record. It specifies (in general terms) what kind of device the driver handles. It also specifies whether the driver is an extended driver.

DIS_driverType
This field has a member of the **DriverType** enumerated type. It specifies specifically what kind of device the driver drives.

The *DIS_driverAttributes* field contains a **DriverAttrs** record. This record has the following fields:

```
DriverAttrs     record
      DA_FILE_SYSTEM:1,
1.2   DA_CHARACTER:1,
      DA_HAS_EXTENDED_INFO:1,
      :13
DriverAttrs     end
```

DA_FILE_SYSTEM
The driver is for file access.

DA_CHARACTER
The driver is for a character-oriented device.

DA_HAS_EXTENDED_INFO
The driver is an extended driver; it provides extra information and extra functionality (described below).

The *DIS_driverType* field contains a member of the **DriverType** enumerated type. This type specifies what kind of driver this is. The type has the following members:

DRIVER_TYPE_VIDEO

DRIVER_TYPE_INPUT

DRIVER_TYPE_MASS_STORAGE

DRIVER_TYPE_STREAM

DRIVER_TYPE_FONT

DRIVER_TYPE_OUTPUT

DRIVER_TYPE_LOCALIZATION

DRIVER_TYPE_FILE_SYSTEM

DRIVER_TYPE_PRINTER

DRIVER_TYPE_SWAP

DRIVER_TYPE_POWER_MANAGEMENT

DRIVER_TYPE_TASK_SWITCH

DRIVER_TYPE_NETWORK

DRIVER_TYPE_SOUND

DRIVER_TYPE_PAGER

DRIVER_TYPE_PCMCIA

DRIVER_TYPE_FEP

DRIVER_TYPE_MAILBOX_DATA

DRIVER_TYPE_MAILBOX_TRANSPORT

DRIVER_TYPE_SOCKET

DRIVER_TYPE_SCAN

DRIVER_TYPE_OTHER_PROCESSOR

DRIVER_TYPE_MAILBOX_RECEIVE

DRIVER_TYPE_MODEM

DRIVER_TYPE_CONNECT_TRANSLATE

DRIVER_TYPE_CONNECT_TRANSFER

Code Display 1-1 A Sample DriverTable

```
;------------------------------------------------------------------------
;       dgroup data
;------------------------------------------------------------------------

idata   segment

        ; First, the driver info structure. Note that we name this structure as
        ; “DriverTable.”

DriverTable     DriverInfoStruct        <
        MyStrategyRoutine,
        <
                0,      ; not a DA_FILE_SYSTEM device driver
                0,      ; not a DA_CHARACTER device
                0,      ; no extended information
        >,
        DRIVER_TYPE_PCMCIA
>
        ; declare the table as public to prevent Esp from generating a warning.

public          DriverTable
```


```
        ; Place any other initialized data here
idata   ends
udata   segment
        ; Place your uninitialized data here
udata   ends
```


1.2.3 Extended Drivers

If the driver is an extended driver (i.e., if the DA_HAS_EXTENDED_INFO bit in the *DIS_driverAttributes* field is set), the device must use a slightly different information structure. Instead of using a **DriverInfoStruct**, it must begin its **dgroup** segment (or fixed, read-only, code resource) with a **DriverExtendedInfoStruct**. The **DriverExtendedInfoStruct** has the following definition:

```
DriverExtendedInfoStruct      struct
      DEIS_common       DriverInfoStruct
      DEIS_resource
                  hptr.DriverExtendedInfoTable
DriverExtendedInfoStruct      ends
```

This structure's first field is a regular **DriverInfoStruct**, so the segment still begins with a **DriverInfoStruct** and a strategy routine, as is required. The other field should contain the handle of a sharable lmem segment that contains the driver's **DriverExtendedInfoTable**.

Extended drivers must have a **DriverExtendedInfoTable** structure. This structure, with its associated data, is generally put in its own resource, a sharable LMem heap. The resource need not (indeed, *should* not) be fixed. The **DriverExtendedInfoTable** structure must be at the beginning of the resource. The **DriverExtendedInfoTable** structure has the following definition:

```
DriverExtendedInfoTable struct
```

```
        DEIT_common       LMemBlockHeader
        DEIT_numDevices   word
        DEIT_nameTable    nptr.lptr.char
        DEIT_infoTable    nptr.word
DriverExtendedInfoTable       ends
```

DEIT_common
: This is the standard LMem block header structure. You must initialize this to "{}". Do not attempt to fill in this field yourself; Esp will fill in this field appropriately.

DEIT_numDevices
: This is the number of different devices supported by this driver.

DEIT_nameTable
: This is a near pointer to an array of chunk handles. Each chunk handle is the handle of a chunk containing the name of a supported device as a null-terminated string. There must be *DEIT_numDevices* different entries in the table.

DEIT_infoTable
: This field contains a near pointer to an array of words. Each word contains driver-specific information for each device. The nature of this information depends on what kind of device driver this is; the data kept in this word is discussed in Code Display 1-2.

For example, suppose you are writing an extended driver that supports three different sound cards. You might set up your driver's informational structures like this:

Code Display 1-2 A Driver's Informational Structures

```
;------------------------------------------------------------------------
;       dgroup data
;------------------------------------------------------------------------

idata segment

; First, the driver info structure. This is an extended driver, so we use the
; DriverExtendedInfoStruct:
```

```
DriverTable     DriverExtendedInfoStruct <
                        <MySoundStrategy,               ; the strategy routine
                         mask DA_HAS_EXTENDED_INFO,;    ; the DriverAttrs record
                         DRIVER_TYPE_SOUND>,            ; The DriverType
                        MySoundExtendedInfoSegment>

idata ends

;------------------------------------------------------------------------
;       Extended info segment
;------------------------------------------------------------------------

MySoundExtendedInfoSegment      segment lmem LMEM_TYPE_GENERAL

; First, the DriverExtendedInfoTable. This must be at the beginning of the
; resource.

MySoundExtendedDriverInfoTable  DriverExtendedInfoTable <
                        {},             ; The LMemBlockHeader;
                                        ; Esp will fill this in
                        length MySoundBoardNames,
                                        ; The number of boards supported
                        offset MySoundBoardNames,
                                        ; near-pointer to table of chunk handles
                        offset MySoundBoardInfoTable
                                        ; near-pointer to table of data words
                        >

; Now, a table of chunk handles. The chunks contain the names of the different
; boards supported.

MySoundBoardNames       lptr.char       FooSound1_0,
                                        FooSound2_0,
                                        Knockoff1_2

                        lptr.char       0

; Now, the names themselves.

LocalDefString  FooSound1_0     <'FooCo Soundarama 1.0', 0>
                                        ; The string must be null-terminated
LocalDefString  FooSound2_0     <'FooCo Soundarama 2.0', 0>
LocalDefString  Knockoff1_2     <'KnockOff SoundClone 1.2', 0>

; And the data words.

MySoundBoardInfoTable   word
                SoundWordOfData         <1,1,1,>,
                SoundWordOfData         <1,1,1,>,
                SoundWordOfData         <1,1,1,>
```


```
MySoundExtendedInfoSegment ends
```

The drivers for some kinds of devices must be extended drivers. Furthermore, some devices require you to use a certain special **InfoStruct**, the first field of which is a **DriverExtendedInfoStruct** or **DriverInfoStruct**. For example, if you are writing a mouse driver, you must begin its driver table segment with a **MouseDriverInfoStruct**, the first field of which is a **DriverExtendedInfoStruct**.

1.2.4 The Strategy Routine

Every driver must have a strategy routine. This routine is called by the GEOS kernel and by libraries. The strategy routine is passed a code telling it what it should do. The strategy routine acts accordingly. Generally, the strategy routine contains a jump table; it calls a different routine, using the passed code as an offset into the jump table. (All the passed codes are even numbers, to facilitate jumping through a table of near-pointers.)

1.2.4.1 What Functions Must Be Handled?

`DriverFunction, DriverExtendedFunction`

The strategy routine must be in a fixed resource. As noted above in "Information about the Driver" on page 13, you should put a pointer to the routine in the driver's **DriverInfoStruct**.

The strategy routine is always passed at least one argument, in the **`di`** register. This argument value specifies what the strategy routine should do. (Other arguments may be passed, depending on what is in **`di`**; the return value also depends on the passed value of **`di`**.) **`di`** may contain one of the following four things:

- A member of the **DriverFunction** enumerated type, the most elemental of all driver functions. This type contains four values: DR_INIT, DR_EXIT, DR_SUSPEND, and DR_UNSUSPEND. All drivers must be able to handle these four functions. (The **DriverFunction** type is discussed below in "The DriverFunction Type" on page 20.)

- A member of the **DriverExtendedFunction** enumerated type. These will only be sent to extended drivers. This type contains two values: DRE_TEST_DEVICE and DRE_SET_DEVICE. All extended drivers must be able to handle these functions. (The **DriverExtendedFunction** type is discussed below in "The DriverExtendedFunction Type" on page 24.)
- A function type specific to the kind of device-driver this is. (For example, PCMCIA drivers should handle the **PCMCIAFunction** codes.) Different types of drivers, of course, need to handle different functions. Mouse drivers, for example, have to handle different functions than print drivers do. The device-specific codes are discussed in the chapter relating to those drivers.

1.2

- An escape code. If the high bit of **di** is set, an escape code is being sent. Different drivers will react to this in different ways. Some drivers will not have to handle escape codes at all.

1.2.4.2 The DriverFunction Type

DR_INIT, DR_EXIT, DR_SUSPEND, DR_UNSUSPEND

Every driver, of whatever type, must handle the four functions specified by the **DriverFunction** type. Even if a device driver wishes to do nothing upon receipt of an event, it must at least handle the function code itself. These functions are bound to the even integers from zero to six. Each of these functions has its own pass and return conventions.

■ DR_INIT

This is sent to the driver when it is first loaded. Typically, a driver will set up whatever interrupt handlers it may have. You might also wish to load any state variables that the driver needs

Pass:	**di**	DR_INIT (= 0).
	cx	value of **di** passed to **GeodeLoad**. If the driver was not loaded through **GeodeLoad**, the value in this register is undefined.
	dx	value of **bp** passed to **GeodeLoad**. If the driver was not loaded through **GeodeLoad**, the value in this register is undefined.
Returns:	CF	Set if initialization failed; the system will then automatically unload the driver.

Destroyed: Allowed to destroy **ax**, **cx**, **dx**, **ds**, **es**, **di**, **si**, **bp**

Include: **driver.def**

DR_EXIT

This is sent to the driver when it is being unloaded. Typically, drivers unregister any interrupt handlers they may have set up.

If the driver is a system driver (i.e., *system* is set within its geode parameters file) then the handler for this function, and any information that handler needs, *must* reside in fixed memory. This allows the driver to be unloaded at the last possible moment.

Pass: **di** DR_EXIT (= 2).

Returns: Nothing.

Destroyed: Allowed to destroy **ax**, **bx**, **cx**, **dx**, **ds**, **es**, **di**, **si**.

Include: **driver.def**

DR_SUSPEND

This is sent to the driver if GEOS is attempting to task-switch out. The driver may refuse to suspend itself.

Pass: **di** DR_SUSPEND (=4).
cx:dx Pointer to a buffer of length DRIVER_SUSPEND_ERROR_BUFFER_SIZE (defined in **driver.def** as 128 bytes).

Returns: CF Set if the driver refuses to suspend. The driver should then write a null-terminated explanatory message, using the standard GEOS character set, to the buffer pointed to by **cx:dx**.

Destroyed: Allowed to destroy **ax**, **di**.

Include: **driver.def**

DR_UNSUSPEND

This is sent to the driver if GEOS is being task-switched back into memory.

Pass: **di** DR_UNSUSPEND (=6).

Returns: Nothing.

Destroyed: Allowed to destroy **ax**, **di**.

1.2.4.3 Writing the Strategy Routine

As noted, the strategy routine is the single entry point upon which a driver executes code. That routine determines what the driver needs to do, and calls the appropriate function from that point.

Code Display 1-3 A Sample Strategy Routine

```
1.2 DefPFunction    macro   routine, constant
    .assert         ($-pfuncs) eq constant*2,   <Routine is not in the right slot!>
    .assert         (type routine eq far),      <Routine is not declared far!>
                    fptr.far        routine
                    endm

    Resident        segment         resource

    pfuncs          label   fptr.far
            ;
            ;Handle the basic four DriverFunction types
            ;
    DefPFunction    MyInit,                 DR_INIT
    DeFPFunction    MyExit,                 DR_EXIT
    DefPFunction    MySuspend,              DR_SUSPEND
    DefPFunction    MyUnsuspend,            DR_UNSUSPEND
            ;
            ; If this is an extended driver, they would appear here
            ; Otherwise, begin the enumerations peculiar to this driver
            ;
    DefPFunction    MyCustomRoutine         DR_MYDRIVER_CUSTOM_ROUTINE

            ;
            ; Write the strategy routine itself
            ;

    MyStrategy      proc    far
                    uses    ds, es
                    .enter

            ; Make sure we can handle the function

                    cmp     di, MyFunction
                    jae     fail
                    test    di, 1   ; check whether the function code is odd (invalid)
                    jnz     fail
```

```
        ; Now call the appropriate driver routine. Load DS and ES with our dgroup
        ; for future use

                segmov  ds, dgroup, ax
                mov     es, ax
                shl     di
                pushdw  cs:[pfuncs][di]
                call    PROCCALLFIXEDORMOVABLE_PASCAL

done:

                .leave
                ret

fail:

                stc             ; set carry if we can't support
                jmp     done

MyStrategy      endp

MyDoNothing     proc    far

                clc
                ret

MyDoNothing     endp

Resident        ends

Init    segment resource

MyInit          proc    far

        ; Handle DR_INIT

MyInit          endp

MyExit          proc    far

        ; Handle DR_EXIT

MyExit          endp

Init    ends
```


1.2.4.4 The DriverExtendedFunction Type

`DRE_TEST_DEVICE, DRE_SET_DEVICE, DevicePresent, EnumerateDevice`

All extended drivers must be able to handle the two functions specified by the **DriverExtendedFunction** type. These are defined in the file **driver.def**. Because these types are enumerated following the **DriverFunction** types, they contain a value of either 8 or 10.

1.2

This file also provides a useful macro for extended drivers, **EnumerateDevice**. This macro locks the block containing the extended driver info, and searches through the name table for the device string passed. This is very useful for handling functions.

■ DRE_TEST_DEVICE

This function instructs the driver to test whether the device needing to be driven is one which is actually able to run, and is present on the system. The null-terminated string name of the device is passed. The strategy routine should return a member of the **DevicePresent** enumerated type. There are four possible return values:

DP_NOT_PRESENT
: Driver knows that the device is not there.

DP_CANT_TELL
: Driver isn't sure whether the device is there.

DP_PRESENT
: Driver knows that the device is there.

DP_INVALID_DEVICE
: The string passed does not contain the name of a device supported by the driver.

Pass:	**di**	DRE_TEST_DEVICE (= 8).
	dx:si	Pointer to null-terminated string containing the name of the device.
Returns:	**ax**	A member of the **DevicePresent** enumerated type.
	CF	Set if **ax** = DP_INVALID_DEVICE, clear otherwise.

Tips & Tricks: The **EnumerateDevice** macro is useful for checking if the string is the name of a supported device. You may use the returned table index to reference another table of test routines.

■ DRE_SET_DEVICE

This function informs the driver which of its devices it is to support.

Pass: **di** DRE_SET_DEVICE (= 10).

Pass: **dx:si** Pointer to a null-terminated string containing the name of the device.

Returns: Nothing.

Destroyed: Allowed to destroy **di**.

Tips & Tricks: The **EnumerateDevice** macro is useful for checking if the string is the name of a supported device. You may use the returned table index to reference another table of test routines.

■ EnumerateDevice

```
EnumerateDevice <infoRes>
```

This macro checks if a string contains the name of a device supported by the driver. If it does, the macro locks the resource containing the driver's extended information.

Pass: **infoRes** Name of the resource containing the driver's extended information.

dx:si Pointer to null-terminated string containing name of device.

Returns: CF Clear if passed string matches name of device supported by the driver; set otherwise.

ax If CF is set, **ax** contains DP_INVALID_DEVICE; otherwise, **ax** is destroyed.

bx Handle of resource containing extended information.

es If CF is clear, **es** contains segment address of locked block containing extended information; otherwise, **es** is destroyed.

di If CF is clear, **di** contains the device's place in the driver's information table. The first device has a "place" of zero, the next device is two, the next is four, etc.
If CF is set, **di** is destroyed.

Destroyed: **cx**, **ds**

Warning: If the macro succeeds in matching the string to a device, it will lock the block containing the driver's extended information and return the block's segment address in **es**. Be sure to unlock this block when you're done with it.

1.2.5 Escape Codes

`DriverEscCode`

Some kinds of drivers may be passed *escape codes*. An escape code is passed to the strategy routine in **`di`**, just like a function code. All escape codes have the sign bit set; they can thus be easily distinguished from other function codes, which have the sign bit cleared.

1.2

```
DriverEscCode     etype word, 8000h, 1
```

How a driver responds to an escape code depends on what kind of device the driver controls. Each kind of device has its own conventions for handling escape sequences, if it handles them at all.

Mouse Drivers

2

Driver Development

Most GEOS platforms will use some kind of pointing device. On desktop machines, this is most commonly a mouse; in any event, these pointing devices share many similarities with mice. Accordingly, all these devices are driven by drivers known collectively as *mouse drivers*.

2.1 Mouse Drivers

Most mouse drivers behave in very similar ways. For this reason, most GEOS mouse drivers share a lot of code. This code is provided in the SDK in the files \OMNIGO\DRIVER\DDK\MOUSE\MOUSECOM.ASM and ...\MOUSE\MOUSESER.ASM. That directory also contains several GEOS mouse drivers; these demonstrate how the drivers actually use the common code to perform such tasks as send mouse movements to the system, handle strategy-routine requests, etc.

2.1.1 Data Structures

All mouse drivers must be extended drivers, even if they support only one kind of mouse. A mouse driver's **dgroup** segment must begin with the **MouseDriverInfoStruct** structure. This structure is based on the **DriverExtendedInfoStruct** structure, but has some extra fields:

```
MouseDriverInfoStruct   struct
      MDIS_common DriverExtendedInfoStruct
            <<0, mask DA_HAS_EXTENDED_INFO,
                  DRIVER_TYPE_INPUT>,
              0>
      MDIS_numButtons   word                  ?
      MDIS_xRes         word                  ?
      MDIS_yRes         word                  ?
      MDIS_flags        MouseDriverInfoFlags  0
MouseDriverInfoStruct   ends
```

MDIS_numButtons
This is the number of buttons the supported mouse has.

MDIS_xRes, *MDIS_yRes*
This is the number of points per inch, of the points collected by the pointing device. Mouse drivers generally have these set to zero; the fields are used for other input devices that use mouse drivers, such as pen-screens.

2.1

MDIS_flags A record of **MouseDriverInfoFlags**. These flags store miscellaneous information about the mouse.

Each **MouseDriverInfoStruct** stores a word of **MouseDriverInfoFlags**. This record has only a single flag:

MDIF_KEYBOARD_ONLY
This driver is actually a keyboard-driven mouse driver, i.e. the user doesn't have a real mouse.

Every mouse driver must set up an extended information resource, as described above in "Driver Basics," Chapter 1. This resource must contain a **DriverExtendedInfoTable**, which (among other things) contains a pointer to an array of data words, one word for each supported mouse. These data words must contain a **MouseExtendedInfo** record. This record has the following flags:

MEI_SERIAL Set if the device is a serial mouse, and needs a COM port to operate.

MEI_GENERIC
Set if this is a generic mouse and needs a DOS-level driver.

MEI_IRQ This field is four bits wide. If it is set, the mouse needs to be told at what interrupt level it is operating (i.e. it is a "bus" mouse). This field should contain the factory-set default value.

MEI_CALIBRATE
Set if this mouse can be calibrated within GEOS.

2.1.2 Functions

Mouse drivers must be able to handle all four functions defined by **DriverFunction**, and both functions defined by

DriverExtendedFunction. Furthermore, they must be able to handle the functions defined by **MouseFunction**, a special enumerated type defined in **mousedr.def**.

As usual, the first of these function names is an enumerated equal to 12 (or two past the last **DriverExtendedFunction**), and the constants increase by two thereafter.

DR_MOUSE_SET_RATE

2.1

The mouse should set the number of times it reports per second.

Pass:	**cx**	The report rate the mouse should be set to, in number of reports per second.
Returns:	**cx**	The actual new report rate for the mouse, again in number of reports per second.

Destroyed: Allowed to destroy **di** and **ax**.

Include: **mousedr.def**

DR_MOUSE_SET_ACCELERATION

The mouse should set its acceleration rate.

Pass:	**cx**	The threshold for acceleration (i.e. if the mouse moves this many pixels in 1/30 second, acceleration should start).
	dx	Acceleration multiplier once threshold is met.
Returns:	Nothing.	

Destroyed: Nothing.

Include: **mousedr.def**

DR_MOUSE_GET_ACCELERATION

The mouse should return its current acceleration rate.

Pass:	Nothing.	
Returns:	**cx**	The threshold for acceleration (i.e. if the mouse moves this many pixels in 1/30 second, acceleration should start).
	dx	Acceleration multiplier once threshold is met.

Destroyed: Nothing.

Include: **mousedr.def**

DR_MOUSE_COMBINE_MODE

The mouse should set the mode for combining mouse events. This is a member of the (byte-sized) **MouseCombineMode** enumerated type:

MCM_ COMBINE

MCM_NO_COMBINE

MCM_COMBINE_COLINEAR_ONLY

2.1

Pass: `cl` **MouseCombineMode** to use.

Returns: Nothing.

Destroyed: Nothing.

Include: **mousedr.def**

DR_MOUSE_GET_COMBINE_MODE

The mouse should return the mode it uses for combining mouse events. This is a member of the (byte-sized) **MouseCombineMode** enumerated type, described above on page 32.

Pass: Nothing.

Returns: `cl` Current **MouseCombineMode**.

Destroyed: Nothing.

DR_MOUSE_GET_CALIBRATION_POINTS

This instructs the mouse driver to return its current set of calibration points.

Pass: `dx:si` Buffer to which to write calibration points. This buffer will be long enough to hold nine i.e. MAX_NUM_CALIBRATION_POINTS) calibration points.

Returns: `dx:si` Pointer to same buffer, filled with calibration points
`cx` Number of calibration points

Destroyed: Nothing

DR_MOUSE_SET_CALIBRATION_POINTS

This instructs the mouse driver to set its calibration points.

Pass: `dx:si` Buffer filled with adjusted calibration points.
`cx` Number of calibration points.

Returns: Nothing.

Destroyed: Nothing.

DR_MOUSE_GET_RAW_COORDINATE

This instructs the mouse driver to return the current calibrated and non-calibrated mouse positions.

Pass: Nothing.

Returns:

CF	Clear if point returned, set otherwise.
(**ax,bx**)	Current raw (uncalibrated) mouse position, if CF = 0.
(**cx,dx**)	Current adjusted (calibrated) mouse position, if CF = 0.

2.1

Destroyed: Nothing.

2.1

Driver Developmer

PCMCIA Drivers

3

Driver Development

This chapter explains the fundamentals associated with writing a PCMCIA driver for GEOS. Life for a PCMCIA driver— as with a human being— begins and ends with traumatic events. The driver's birth begins with the insertion of a card into the device; its death is marked by the removal of that card. When writing a PCMCIA driver, you may want to contemplate your own end and handle the removal case elegantly, for the driver's sake if not for your own.

When a card is first inserted into a PCMCIA "socket" (or "slot") GEOS will load all drivers for that card that it finds, one after the other. Those that are found to be compatible with the card remain loaded as long as the card remains within the chosen socket. The removal of a card is the other big event in the PCMCIA driver's life. At that point, if anything is actively using the card, the driver can raise an objection to that removal and attempt to resolve any conflicts that arise.

A PCMCIA driver is a complex driver; there are many other events besides insertion and removal that a typical driver will need to handle. These events are usually functions of CardServices, a third-party library licensed to Geoworks, and, more specifically, the GEOS PCMCIA library interface to CardServices. This chapter is not meant to document all of these complex cases because drivers, whether file system or serial, may exhibit vastly different characteristrics. You will want to consult the **pcmcia.def** file on the SDK for more information on handling these functions.

A PCMCIA sample driver that you may use as a template is available in \OMNIGO\DRIVER\DDK\PCMCIA\SAMPLE. Other (functioning) drivers are located within \OMNIGO\DRIVER\DDK\PCMCIA.

3.1 PCMCIA Drivers

Writing a GEOS device driver for PCMCIA cards is somewhat different than writing a driver for other devices. The driver acts not only with the device, but with the PCMCIA library (a GEOS library) and CardServices.

A PCMCIA driver must handle the four basic **DriverFunction** calls and must also handle the specific **PCMCIAFunction** calls defined in

pcmciaDr.def. PCMCIA drivers are not extended, so they do not need to handle the **DriverExtendedFunction** calls.

Because your driver must interact with CardServices, it must also define a callback routine handling the specific **CardServicesEventCode** types. It will also need to send function calls to CardServices using the **CardServicesFunction** type.

3.1

3.1.1 State Information

A PCMCIA driver needs to maintain information about the card(s) and socket(s) that it is driving. The "socket" refers to the physical slot of the PCMCIA hardware interface. (This should not be confused with the GEOS Socket library API.) Each driver should contain the capability to support multiple sockets since hardware platforms may contain multiple PCMCIA ports. Also, CardServices may map multiple logical sockets to a single physical socket; this allows CardServices to mimic multi-function cards.

A driver should keep information about each socket in some structure table. The information stored in this table is, of course, up to the driver. Examples of information that may be necessary:

- The socket number.
- Whether the card was removed while it was in use.
- How the card was configured. This depends on the requirements of the device.

3.1.2 Handling Basic Functions

`DR_INIT, DR_EXIT`

Your PCMCIA driver will need to handle the basic DR_INIT and DR_EXIT routines defined in **driver.def**. (There usually is not any need to handle DR_SUSPEND and DR_UNSUSPEND.) Because PCMCIA drivers are not extended, there is no need to handle DRE_TEST_DEVICE and DRE_SET_DEVICE.

3.1.2.1 Insertion

As noted, the insertion or removal of a PCMCIA card are the two most important events in the driver's life. When a driver is first loaded, it registers with CardServices. Among other things, this registration allows the driver to be told when a card is inserted. (Be patient; this is not as non-sensical as it seems.)

Of course, inserting a card prompted the registration in the first place! This only means that the driver is guaranteed to receive notification that at least one card was inserted, after registration with CardServices is complete. Once informed of this event (or additional insertion events), the driver must examine the card (now within a "socket") to discern whether it can support the card. If the card is compatible, the driver configures the card according to its own specifications and makes its devices and/or memory available to GEOS.

3.1.2.2 Removal

The removal of a PCMCIA card is a difficult event for a PCMCIA driver. The driver may be writing a file to the card; it may be communicating something over the serial line. No one wants to "go" when confronted with the tasks still remaining, but unlike us, a PCMCIA driver has the option of objecting to its removal.

If something is actively using the card, the driver must tell the PCMCIA library to object to the card's removal. The library will inform the user that the removal was effected under hostile protest. The user then has the option of reinserting the card and either leaving it in (and allowing whatever objected to the removal finish its business) or force GEOS to stop accessing the card (for example, by closing applications). The user always has the option of rebooting the system if the card is no longer available.

The last thing a driver will do is unregister itself with CardServices. Afterward, the socket is closed and the driver is unloaded.

DR_INIT

This function is sent to the driver by the kernel when the driver is first loaded. A PCMCIA device driver should handle this call by registering as a

CardServices client. It does this by invoking the **CardServicesFunction** CSF_REGISTER_CLIENT event. (See "PCMCIA Library Functions" on page 49.) This registration is a separate issue than registration with the PCMCIA library itself; that should take place in your DR_PCMCIA_CHECK_SOCKET routine. (At that time, the driver will know into which socket the card was inserted.)

3.1

The driver should return carry set (failure) only if it is incompatible with CardServices or the current environment in some way. The driver cannot consult the card to see if it supports it yet; it does not yet know into what socket the card was inserted. (That information is provided at a later time by the CSEC_CARD_INSERTION event from CardServices.)

This function is guaranteed to occur before a DR_PCMCIA_CHECK_SOCKET event; that function must wait for the registration (initiated by this handler) to be complete before checking whether the card is supported.

Pass:	**di**	DR_INIT (= 0).
	cx	value of **di** passed to **GeodeLoad**. If the driver was not loaded through **GeodeLoad**, the value in this register is undefined.
	dx	value of **bp** passed to **GeodeLoad**. If the driver was not loaded through **GeodeLoad**, the value in this register is undefined.
Returns:	CF	Set if initialization failed; the system will then automatically unload the driver.

Destroyed: Allowed to destroy **ax**, **cx**, **dx**, **ds**, **es**, **di**, **si**, **bp**

Include: **driver.def**

Code Display 3-1 Sample DR_INIT Routine

```
SampleInit      proc    far
        .enter

        ; If you will need another driver to get your work done, retrieve its
        ; strategy routine here and store it. Fetching the strategy routine of
        ; another driver involves loading in the core block, which we can't do at
        ; interrupt time.

        ; Register as a CardServices client. We do this by retrieving the address
        ; of the callback routine (which must be in fixed memory) and sending
        ; CSF_REGISTER_CLIENT to CardServices.
```

```
        mov     di, segment     SampCardServicesCallback
        mov     si, offset      SampCardServicesCallback
        mov     cx, size regArgList
        segmov  es, cs
        mov     bx, offset regArgList
        CallCS  CSF_REGISTER_CLIENT
        jc      fail

        ;
        ; ds is assumed loaded by the strategy routine ...
        ;                                                              3.1

        mov     ds:[csHandle], CS_HANDLE_REG
        clc

done:
        .leave
        ret

fail:
        stc
        jmp     done

SampleInit      endp

        ;
        ; Set this thing up appropriate to the card you're driving.
        ;
        ; In some cases, your driver may manage an I/O device but register as
        ; a memory client to make sure the automatic configuration client
        ; gets called to configure the card before this driver gets called.
        ; It depends on who does the actual configuration for the cards you
        ; manage.
        ;

regArgList      CSRegisterClientArgs <
        mask CSRCAA_ARTIFICIAL_EXCLUSIVE or mask CSRCAA_ARTIFICIAL_SHARED or \
                mask CSRCAA_MCD,
        mask CSEM_CARD_DETECT_CHANGE,
        < 0, segment dgroup, 0, 0>,
        0201h
>

Init    ends
```


■ DR_EXIT

This function is sent to a driver by the kernel when it is being unloaded. A PCMCIA device driver should handle this call by at least de-registering as a CardServices client. It does this by sending the CSF_DEREGISTER_CLIENT event defined in the PCMCIA library. The driver should also release any resources it may have requested from CardServices.

3.1

Pass: **di** DR_EXIT (= 2).

Returns: Nothing.

Destroyed: Allowed to destroy **ax**, **bx**, **cx**, **dx**, **ds**, **es**, **di**, **si**.

Include: **driver.def**

Code Display 3-2 Sample DR_EXIT Routine

```
SampleExit      proc    far
                uses    ax, cx, dx
        .enter

        clr     cx
        mov     dx, ds:[csHandle]
        CallCS  CSF_DEREGISTER_CLIENT

        clc
        .leave
        ret

SampleExit       endp
```

3.1.3 PCMCIA Driver Functions

DR_PCMCIA_CHECK_SOCKET, DR_PCMCIA_OBJECTION_RESOLVED, DR_PCMCIA_CLOSE_SOCKET, DR_PCMCIA_DEVICE_ON, DR_PCMCIA_DEVICE_OFF

In addition to handling the basic functions, a PCMCIA driver must be able to handle the functions defined by **PCMCIAFunction**, a special enumerated type defined in **pcmciaDr.def**.

The first of these function names is an enumerated value equal to 8 (or two past the last **DriverFunction**), and the constants increase by two thereafter.

DR_PCMCIA_CHECK_SOCKET

The PCMCIA library calls this function when a card has been inserted into the device. This function will only occur after a DR_INIT has already occurred; that function should register the socket as a CardServices client.

3.1

In your handler for this function, make sure to wait for the registration with CardServices to complete. (This is typically done with a wait loop; the loop checks the state of a driver flag indicating whether notification has been completed.) After this has occurred, the PCMCIA driver should also check the socket for compatibility with the card inserted.

This function typically takes place after a CSEC_CARD_INSERTION event has occurred. Drivers should respond to a CSEC_CARD_INSERTION event by setting a driver flag indicating whether the indicated driver supports the card.

A driver responding to a DR_PCMCIA_CHECK_SOCKET should wait until registration is complete (by receipt of a CSEC_REGISTRATION_COMPLETE event). At that point, they can test whether the CSEC_CARD_INSERTION event occurred smoothly.

If the card is supported, the driver should then register with the PCMCIA library using **PCMCIARegisterDriver**. This will register your driver with the PCMCIA library for the particular card in the particular socket . Note that this is a separate registration than that with CardServices.

Pass:	**cx**	Socket number.
	di	DR_PCMCIA_CHECK_SOCKET
Returns:	CF	Set if PCMCIA card in the socket is supported by the driver.
Destroyed:	**di**	
Include:	**pcmciaDr.def**	

Code Display 3-3 Sample DR_PCMCIA_CHECK_SOCKET Routine

```
SampIeCheckSocket       proc    far
socket          local   word    push cx
        uses    ax, bx, cx, dx, si
        .enter

        ;
        ; Wait to make sure we've received all the artificial insertion
3.1     ; events so we know whether we support the card.
        ;

waitForRegistrationLoop:
        tst     ds:[amRegistered]
        jz      waitForRegistrationLoop

        ;
        ; See if we support the thing.
        ;

        call    SampUDerefSocket
                CheckHack <SCS_NO eq 0>
        tst     ds:[bx].SSI_support
        jnz     processIt

fail:
        clc     ; not our card

done:
        .leave

ret

processIt:
        ;
        ; If the card has been configured by someone outside of this driver,
        ; here's where you'd fetch the configuration info from Card Services
        ; and tell other people about it so the thing can be used within
        ; GEOS.
        ;

        PrintMessage <INSERT CODE HERE>

        ;
        ; Register with the library, finally.
        ; bx    <- geode handle
        ; cx    <- socket
```

```
        ; dx    <- cs handle
        ; es:di <- CSRegisterClientArgs
        ; ax:si <- cs callback
        ;

        mov     bx, vseg regArgList
        call    MemLockFixedOrMovable
        mov     es, ax
        mov     di, offset regArgList

        mov     bx, handle 0                                    3.1

        mov     cx, ss:[socket]

        segmov  ds, dgroup, ax
        mov     dx, ds:[csHandle]

        mov     ax, segment SampCardServicesCallback
        mov     si, offset SampCardServicesCallback
        call    PCMCIARegisterDriver

        mov     bx, vseg regArgList
        call    MemUnlockFixedOrMovable

        stc                     ; happy happy happy
        jmp     done

SampIeCheckSocket endp
```

DR_PCMCIA_OBJECTION_RESOLVED

The PCMCIA library calls this function when the user has answered an objection raised to the removal of a card. The function passes a **PCMCIAObjectionResolution** type in **dx** indicating the nature of the resolution. If that value is PCMOR_CLEAN_UP, then the user has asked that the card be ejected. The driver should attempt to remove any references to the card.

```
PCMCIAObjectionResolution  etype word, 0, 1
PCMOR_CLEAN_UP             enum PCMCIAObjectionResolution
PCMOR_USER_CANCELED        enum PCMCIAObjectionResolution
PCMOR_SYSTEM_CANCELED      enum PCMCIAObjectionResolution
```

If the user of system cancelled the removal, the driver should simply note that the objection has been resolved.

A driver should always respond to the PCMOR_CLEAN_UP event, even if it did not raise an objection, as the user may not have actually removed the card. This allows the driver to clean up before a card is removed (for example if the user is initiating an ejection of the card through software control).

Pass: **cx** Socket number.
dx **PCMCIAObjectionResolution.**
di DR_PCMCIA_OBJECTION_RESOLVED

3.1 **Returns:** CF (Only meaningful if PCMOR_CLEAN_UP was passed) Clear if the driver was able to remove all references to the card; set otherwise.

Destroyed: **di**

Include: **pcmciaDr.def**

Code Display 3-4 Sample DR_PCMCIA_OBJECTION_RESOLVED Routine

```
SampleObjectionResolved proc     far
        .enter
                CheckHack <PCMOR_CLEAN_UP eq 0>
        tst_clc dx
        jz      attemptCleanUp

        ;
        ; In theory, since the removal was canceled and the card is back, we'd
        ; release any access blocks we might have placed, allowing the card to
        ; be reached again.... we currently set no blocks, though, so this
        ; is a nop.
        ;

        PrintMessage <MAYBE INSERT CODE HERE>

done:
        .leave
        ret

attemptCleanUp:
        ;
        ; Here's where you'd try to get the things that are using the card to
        ; stop using them. Sometimes you can't do that, in which case you
        ; return carry set if the card is still in-use. If you can, though,
        ; try for a bit and occasionally call SampCSCheckCardInUse to see if
        ; the card's still in use.
        ;
```

```
        PrintMessage <INSERT CODE HERE>
        call    SampCSCheckCardInUse
        jmp     done

SampleObjectionResolved endp
```

■ DR_PCMCIA_CLOSE_SOCKET

3.1

The PCMCIA library calls this function when it is about to close a socket; the driver should respond by cleaning up any auxiliary structures created during DR_PCMCIA_CHECK_SOCKET. The PCMCIA library only sends this function if no one has objected to the removal of the card. The driver, at this point, is about to be unloaded.

Pass: **cx** Socket number.
di DR_PCMCIA_CLOSE_SOCKET

Returns: Nothing.

Destroyed: **di**

Include: **pcmciaDr.def**

Code Display 3-5 Handling DR_PCMCIA_CLOSE_SOCKET

```
SampleCloseSocket       proc   far
        uses    ax, di, ds, bx, cx
        .enter

        ;
        ; Here you'd tell the rest of the world that the thing no longer
        ; exists. At this point, we know the card wasn't being used, and
        ; things should have been done in SampleHandleRemoval to ensure that
        ; no one could start using the card after that routine returned.
        ;

        PrintMessage <INSERT CODE HERE>
        .leave
        ret

SampleCloseSocket endp
```

■ DR_PCMCIA_DEVICE_ON

The PCMCIA library calls this function in response to a request (by the power management driver) to turn power on to an indicated socket. This may occur when someone wishes to turn on a socket and the library believes that power is off (for example, after a DR_PCMCIA_DEVICE_OFF function or a CSEC_CARD_INSERTION event). The driver may either call **PCMCIASocketOn** or its own custom function in response to this request.

3.1

Only drivers that invoke the Card Services CSF_REQUEST_CONFIGURATION function will receive this function. (Drivers may also steal configuration ownership through **PCMCIAChangeConfigurationOwner**.)

Pass: `cx` Socket number.
`di` DR_PCMCIA_DEVICE_ON

Returns: Nothing.

Destroyed: `di`

Include: **pcmciaDr.def**

■ DR_PCMCIA_DEVICE_OFF

The PCMCIA library calls this function in response to a request to turn power off to the indicated socket. (The library ensures that a sufficient time elapses without a subsequent request to turn the power on.) The driver may either call **PCMCIASocketOff** or its own custom function in response to this request.

Only drivers that invoke the Card Services CSF_REQUEST_CONFIGURATION function will receive this function. (Drivers may also steal configuration ownership through **PCMCIAChangeConfigurationOwner**.)

Pass: `cx` Socket number.
`di` DR_PCMCIA_DEVICE_OFF

Returns: Nothing.

Destroyed: `di`

Include: **pcmciaDr.def**

3.2 PCMCIA Library Functions

As noted, a PCMCIA driver will interact with both a PCMCIA library and, through that library, CardServices. The PCMCIA library provides a number of routines to aid in communicating with CardServices.

PCMCIARegisterDriver

This routine registers a PCMCIA device driver in the indicated socket. This routine is usually called after the driver is first called with DR_PCMCIA_CHECK_SOCKET for each supported card. 3.2

Pass:

cx	The socket number.
bx	The driver's **GeodeHandle**.
es:di	**CSRegisterClientArgs** passed to CardServices.
ax:si	The CardServices callback routine.
dx	The CardServices client handle for the driver.

Returns: Nothing.

Destroyed: **ax**

Include: **pcmcia.def**

PCMCIAObjectToRemoval

This routine notes a driver's objection to the removal of a card from the device. In calling this routine, the driver is dedicated to wait for a DR_PCMCIA_OBJECTION_RESOLVED function before taking further action with the card.

Pass:

cx	The socket number.
dx	Set (non-zero) if the card is non-removable.
bp	Handle of the driver.

Returns: Nothing.

Destroyed: **ax, di**

Include: **pcmcia.def**

PCMCIAExclusiveGranted

This routine should be called to acknowledge that a driver has received a CSEC_EXCLUSIVE_COMPLETE.

Pass:

bx	Handle of the driver geode.

Returns: Nothing.

Destroyed: Nothing.

Include: **pcmcia.def**

3.3 CardServices Functions

3.3

A driver must contact CardServices through use of **CardServicesFunction** types defined in **pcmcia.def**. Consult that file for a complete list of all possible function calls, as well as pass and return information for those calls.

The following definitions are for those function types that a driver must use in registering and deregistering your driver with CardServices. Registration with CardServices should be accomplished in your driver's DR_INIT handler. Deregistration should be performed within your driver's DR_EXIT handler.

All **CardServicesFunction** types return the carry flag set if they encounter an error and return a **CardServicesReturnCode** in **ax**. These return codes are enumerated below:

```
CardServicesReturnCode      etype word, 0, 1
CSRC_SUCCESS                enum  CardServicesReturnCode
CSRC_BAD_ADATPER            enum  CardServicesReturnCode
                    ; (sic)
CSRC_BAD_ATTRIBUTE          enum  CardServicesReturnCode
CSRC_BAD_BASE               enum  CardServicesReturnCode
CSRC_BAD_EDC                enum  CardServicesReturnCode
CSRC_RESERVED_1             enum  CardServicesReturnCode
CSRC_BAD_IRQ                enum  CardServicesReturnCode
CSRC_BAD_OFFSET             enum  CardServicesReturnCode
CSRC_BAD_PAGE               enum  CardServicesReturnCode
CSRC_READ_FAILURE           enum  CardServicesReturnCode
CSRC_BAD_SIZE               enum  CardServicesReturnCode
CSRC_BAD_SOCKET             enum  CardServicesReturnCode
CSRC_RESERVED_2             enum  CardServicesReturnCode
CSRC_BAD_TYPE               enum  CardServicesReturnCode
CSRC_BAD_VCC                enum  CardServicesReturnCode
CSRC_BAD_VPP                enum  CardServicesReturnCode
```

```
CSRC_RESERVED_3              enum  CardServicesReturnCode
CSRC_BAD_WINDOW              enum  CardServicesReturnCode
CSRC_WRITE_FAILURE           enum  CardServicesReturnCode
CSRC_RESERVED_4              enum  CardServicesReturnCode
CSRC_NO_CARD                 enum  CardServicesReturnCode
CSRC_UNSUPPORTED_FUNCTION    enum  CardServicesReturnCode
CSRC_UNSUPPORTED_MODE        enum  CardServicesReturnCode
CSRC_BAD_SPEED               enum  CardServicesReturnCode
CSRC_BUSY                    enum  CardServicesReturnCode
CSRC_GENERAL_FAILURE         enum  CardServicesReturnCode
CSRC_WRITE_PROTECTED         enum  CardServicesReturnCode
CSRC_BAD_ARG_LENGTH          enum  CardServicesReturnCode
CSRC_BAD_ARGS                enum  CardServicesReturnCode
CSRC_CONFIGURATION_LOCKED    enum  CardServicesReturnCode
CSRC_IN_USE                  enum  CardServicesReturnCode
CSRC_NO_MORE_ITEMS           enum  CardServicesReturnCode
CSRC_OUT_OF_RESOURCE         enum  CardServicesReturnCode
CSRC_BAD_HANDLE              enum  CardServicesReturnCode
```

3.3

■ CSF_REGISTER_CLIENT

This function instructs CardServices to register the driver. This function must be passed a structure of **CSRegisterClientArgs** containing (among other things) the address of the callback routine with which CardServices should contact the driver. CardServices will send **CardServicesEventCode** types to this callback routine. (For more information on defining your callback routine see “CardServices Events” on page 53.)

This registration should occur when the driver is first loaded, upon receipt of DR_INIT.

```
CSRegisterClientArgs            struct
        CSRCA_attributes        CSRegisterClientArgsAttributes
        CSRCA_eventMask         CSEventMask
        CSRCA_clientData        CSClientData
        CSRCA_version           word
CSRegisterClientArgs            ends

CSRegisterClientArgsAttributes  record
        :11
        CSRCAA_ARTIFICIAL_EXCLUSIVE:1    ; want artificial INSERTION events
                                         ; after exclusive access released
        CSRCAA_ARTIFICIAL_SHARED:1       ; want artificial INSERTION events for all
```


```
                                        ; cards resident when client registers
        CSRCAA_IO:1                     ; I/O cards
        CSRCAA_MTD:1                    ; Memory Technology Driver
        CSRCAA_MCD:1                    ; Memory cards
CSRegisterClientArgsAttributes  end

CSEventMask             record
        :5
        CSEM_SOCKET_SERVICES_UPDATED:1
        CSEM_RESET:1
3.3     CSEM_POWER_MANAGEMENT_CHANGE:1
        CSEM_CARD_DETECT_CHANGE:1
        CSEM_READY_CHANGE:1
        CSEM_BATTERY_LOW:1
        CSEM_BATTERY_DEAD:1
        CSEM_INSERTION_REQUEST:1
        CSEM_EJECTION_REQUEST:1
        CSEM_CARD_LOCK_CHANGE:1
        CSEM_WRITE_PROTECT_CHANGE:1
CSEventMask             end

CSClientData    struct
        CSCD_data       word            ; DI for callback
        CSCD_segment    word            ; DS for callback
        CSCD_offset     word            ; SI for callback
        CSCD_extra      word            ; reserved word that's not
                                        ; loaded into anything...
CSClientData    ends
```

Pass:	**al**	CSF_REGISTER_CLIENT
	cx	Argument length
	es:bx	**CSRegisterClientArgs**
	di:si	Entry point (callback routine) of the driver
Returns:	CF	Set if failure
	ax	**CardServicesReturnCode**
	dx	Client handle
Include:	**pcmcia.def**	

■ CSF_DEREGISTER_CLIENT

This function instructs CardServices to deregister the driver. This function must be passed the client handle returned when the driver first registered with CardServices.

This deregistration should occur when the driver is unloaded, upon receipt of DR_EXIT.

Pass:	**al**	CSF_DEREGISTER_CLIENT
	dx	Client handle.
	cx	No arguments.
Returns:	CF	Set if failure
	ax	**CardServicesReturnCode**
Include:	**pcmcia.def**	

3.4

CallCS

```
CallCS   <command, options>
```

This macro issues a call to CardServices. It must be passed a **CardServicesFunction** to invoke.

Due to interrupt timing concerns, if the macro is called from within a CardServices callback procedure (or from a routine that is called by such a procedure), DONT_LOCK_BIOS must be passed as an option. At all other times you must not pass DONT_LOCK_BIOS (unless you call **SysLockBIOS** yourself) as CardServices is not re-entrant.

3.4 CardServices Events

The interface between CardServices and your driver occurs not only through use of the PCMCIA library; your driver must also handle events sent by CardServices as well. This is performed through use of a callback routine. CardServices will send these events using either a timer interrupt or a status-change interrupt (such as a physical card insertion or removal).

When your driver registers with CardServices, it must pass the address of a callback routine. Your driver should respond to **CardServicesEventCode** types sent to this callback routine and return appropriate **CardServicesReturnCode** types.

The **CardServicesEventCode** functions pass the following arguments to your callback routine:

al	**CardServicesEventCode**
cx	Socket number
dx	Info
di	**di** register for callback routine
ds	**ds** register for callback routine
si	**si** register for callback routine
ss	MTD request segment
bp	MTD request offset
es	Buffer segment
bx	Buffer offset or miscellaneous register

3.4

As with any CardServices functions, you should return the carry flag set if you encounter an error and a **CardServicesReturnCode** in **ax**.

■ CSEC_CARD_INSERTION

A driver receives this event when CardServices determines that a card has been inserted in a PCMCIA socket. The driver should respond by configuring its card in whatever way it sees fit. A driver may also receive this event if some other client received exclusive access to the card and is now relinquishing it. This is transparent to the driver receiving this event.

If a driver receives this event while an unresolved objection to a previous removal is currently active, it should reconfigure the card to its previous state before the objection to removal was noted, if possible; it must also wait to release any blocks (containing the "unresolved" information) until a DR_PCMCIA_OBJECTION_RESOLVED function is received.

Code Display 3-6 Sample CSEC_CARD_INSERTION Handler

```
SampleHandleInsertion   proc    near
        uses    bx, dx
        .enter

        ;
        ; Point to our data record for the socket.
        ;

        call    SampUDerefSocket
```

```
        ;
        ; Here's where you'd examine the card's CIS to see if it's something
        ; you support, then attempt to set it to one of its configurations.
        ; If all that succeeds, you'd set ds:[bx].SSI_support to SCS_YES.
        ; If any of that fails, you'd set ds:[bx].SSI_support to SCS_NO.
        ;
        PrintMessage <INSERT CODE HERE>

setYes::
        mov     ds:[bx].SSI_support, SCS_YES                                    3.4

        ;
        ; See if the card was removed under protest.
        ;
        tst     ds:[bx].SSI_conflict
        jz      clearConflict

        ;
        ;
        ; If this card is coming back in after having been removed while
        ; in-use you may need to tell another driver to restore the state of the
        ; card (this is what happens in the CIDSer driver, for example, where the
        ; baud rate and other parameters need to be restored here).
        ;
        ; If you block people's access to the card while it's in conflict,
        ; this is the time to wake them all up using code like this:
        ;
        ;       call    SysEnterCritical
        ;       VAllSem ds, [bx].SSI_conflictSem
        ;       mov     ds:[bx].SSI_conflictSem.Sem_value, 0
        ;       call    SysExitCritical
        ;
        ; The Enter/ExitCritical prevents other threads from running so we can
        ; reliably set the Sem_value to 0 (it ends up at 1) to cause people
        ; to block immediately the next time the card is in conflict.
        ;
        PrintMessage <INSERT CODE HERE>

clearConflict:
        mov     ds:[bx].SSI_conflict, 0

done:
        .leave
        ret
```


```
setNo:
        mov     ds:[bx].SSI_support, SCS_NO
        jmp     done

SampleHandleInsertion endp
```

■ CSEC_CARD_REMOVAL

3.4

A driver receives this event when Card Services determines that the card has been removed. If this removal is acceptable, the driver should release all Card Services-related resources that it had allocated and make sure not to access the card in the future.

A client may also receive this event if another client has been granted exclusive access to the card. When the other driver relinquishes exclusive access, the previously contacted drivers will receive CSEC_CARD_INSERTION events.

If a driver has been granted exclusive access to a card and receives a CSEC_CARD_REMOVAL event, it should call **PCMCIAExclusiveCardRemoved**. If instead, the driver wishes to raise an objection to this card removal (for example, a serial port was in use or a file is currently open on the card) it should call the **PCMCIAObjectToRemoval** library utility routine. The driver must then wait for the objection to be resolved (through a DR_PCMCIA_OBJECTION_RESOLVED event). If the driver receives a fresh CSEC_CARD_INSERTION event, it should reconfigure the card if it is able. It should not grant access to the card until DR_PCMCIA_OBJECTION_RESOLVED is received.

Code Display 3-7 Sample CSEC_CARD_REMOVAL Handler

```
SampleHandleRemoval     proc    near
        uses    bx, dx, bp
        .enter

        call    SampUDerefSocket

        ;
        ; If card not supported, we're happy to see it go (why are we here?)
        ;
```

```
        tst     ds:[bx].SSI_support
        jz      resetSupport

        ;
        ; See if the card is being used by GEOS using whatever means are
        ; available/appropriate.
        ;

        PrintMessage <INSERT CODE HERE>
        call    SampCSCheckInUse
        jne     resetSupport                                              3.4

        ;
        ; It is in-use, so mark the port conflicted and tell the PCMCIA library
        ; of our objections.
        ;

        mov     ds:[bx].SSI_conflict, TRUE

        mov     dx, TRUE    ; card may NOT be removed
        mov     bp, handle 0
        call    PCMCIAObjectToRemoval

resetSupport:
        ;
        ; Always set SSI_support back to SCS_NO, as it reflects our opinion of
        ; the current state of the socket.
        ;

        mov     ds:[bx].SSI_support, SCS_NO
        clc
        mov     ax, CSRC_SUCCESS
        .leave
        ret

SampleHandleRemoval endp
```

CSEC_EXCLUSIVE_COMPLETE

A driver receives this event when CardServices grants a client driver exclusive access to the PCMCIA socket. The driver should acknowledge that it has received the event by calling **PCMCIAExclusiveGranted**.

■ CSEC_EXCLUSIVE_REQUEST

A driver receives this event when CardServices requests, at the behest of another driver, exclusive access to the card. The driver should react negatively to this event if it objects to this exclusive access. The criteria for this objection should be much the same as if it had received a CSEC_CARD_REMOVAL event.

3.4

Code Display 3-8 Sample CSEC_EXCLUSIVE_REQUEST Handler

```
SampCSHandleExclusiveRequest    proc     near
        uses    bx, di
        .enter

        call    SampCSCheckCardInUse
        jnc     done

        ;
        ; Card is in use - don't allow the exclusive access.
        ;
        mov     ax, CSRC_IN_USE

        stc

done:
        .leave
        ret

SampCSHandleExclusiveRequest endp
```

■ CSEC_CLIENT_INFO

A driver receives this event when CardServices requests standard client information.

Code Display 3-9 Sample CSEC_CLIENT_INFO Handler

```
        ;
        ; Remember that this is not a complete routine.
```

```
doInfo:
        test    es:[bx].CSGCIA_attributes, mask CSGCIAA_INFO_SUBFUNCTION
        jnz     unsupported ; only handle function 0

        ;
        ; Return info about this client to whomever is asking.
        ;

        mov     cx, cs:[clientInfo].CSGCIA_infoLen
        cmp     cx, es:[bx].CSGCIA_maxLen
        jbe     copyInfo                                                    3.4
        mov     cx, es:[bx].CSGCIA_maxLen

copyInfo:
        segmov ds, cs
        mov     si, offset clientInfo.CSGCIA_infoLen
        lea     di, es:[bx].CSGCIA_infoLen
        sub     cx, offset CSGCIA_infoLen       ; not copying all stuff
                                                ; up to here

        rep     movsb
        jmp     success
```

The following is a complete list of **CardServicesEventCode** routines defined in **pcmcia.def**.

CardServicesEventCode	etype	word
CSEC_PM_BATTERY_DEAD	(001h)	
CSEC_PM_BATTERY_LOW	(002h)	
CSEC_CARD_LOCK	(003h)	
CSEC_CARD_READY	(004h)	
CSEC_CARD_REMOVAL	(005h)	
CSEC_CARD_UNLOCK	(006h	
CSEC_EJECTION_COMPLETE	(007h)	
CSEC_EJECTION_REQUEST	(008h)	
CSEC_INSERTION_COMPLETE	(009h)	
CSEC_INSERTION_REQUEST	(00ah)	
CSEC_PM_RESUME	(00bh)	
CSEC_PM_SUSPEND	(00ch)	
CSEC_EXCLUSIVE_COMPLETE	(00dh)	
CSEC_EXCLUSIVE_REQUEST	(00eh)	
CSEC_RESET_PHYSICAL	(00fh)	

	CSEC_RESET_REQUEST	(010h)
	CSEC_CARD_RESET	(011h)
	CSEC_MTD_REQUEST	(012h)
	CSEC_RESERVED_1	(013h)
	CSEC_CLIENT_INFO	(014h)
	CSEC_TIMER_EXPIRED	(015h)
	CSEC_SS_UPDATED	(016h)
3.4	CSEC_CARD_INSERTION	(040h)
	CSEC_RESET_COMPLETE	(080h)
	CSEC_ERASE_COMPLETE	(081h)
	CSEC_REGISTRATION_COMPLETE	(082h)

Your driver will need to create a table to map these event codes to the handlers to invoke for each.

Note that the CSEC_CARD_INSERTION, CSEC_RESET_COMPLETE, CSEC_ERASE_COMPLETE and CSEC_REGISTRATION_COMPLETE events do not follow the simple incremental numbering of the previous events. You will need to check for these events individually, rather than through a simple jump table.

Your handler should respond with an appropriate **CardServicesReturnCode**.

For example, the sample PCMCIA driver included on the SDK defines the following table:

Code Display 3-10 A Sample CardServices Event Table

```
        ; It is usually convenient to define such a table within the Callback
        ; routine itself.

DefCSEvent      macro   event, handler
        .assert ($-eventRoutineTable)/2 eq (event-1)
        nptr.near       handler
                endm

eventRoutineTable       label   nptr
DefCSEvent      CSEC_PM_BATTERY_DEAD,           doIgnore
DefCSEvent      CSEC_PM_BATTERY_LOW,            doIgnore
DefCSEvent      CSEC_CARD_LOCK,                 doIgnore
```

```
DefCSEvent      CSEC_CARD_READY,                doIgnore
DefCSEvent      CSEC_CARD_REMOVAL,              doRemoval
DefCSEvent      CSEC_CARD_UNLOCK,               doIgnore
DefCSEvent      CSEC_EJECTION_COMPLETE,         doIgnore
DefCSEvent      CSEC_EJECTION_REQUEST,          doIgnore
DefCSEvent      CSEC_INSERTION_COMPLETE,        doIgnore
DefCSEvent      CSEC_INSERTION_REQUEST,         doIgnore
DefCSEvent      CSEC_PM_RESUME,                 doIgnore
DefCSEvent      CSEC_PM_SUSPEND,                doIgnore
DefCSEvent      CSEC_EXCLUSIVE_COMPLETE,        doIgnore
DefCSEvent      CSEC_EXCLUSIVE_REQUEST,         doExclusiveReq
DefCSEvent      CSEC_RESET_PHYSICAL,            doIgnore
DefCSEvent      CSEC_RESET_REQUEST,             doIgnore
DefCSEvent      CSEC_CARD_RESET,                doIgnore
DefCSEvent      CSEC_MTD_REQUEST,               unsupported
DefCSEvent      CSEC_RESERVED_1,                unsupported
DefCSEvent      CSEC_CLIENT_INFO,               doInfo
DefCSEvent      CSEC_TIMER_EXPIRED,             doIgnore
DefCSEvent      CSEC_SS_UPDATED,                doIgnore
endEventRoutineTable    label   nptr
```

Code Display 3-11 A Sample PCMCIA CardServices Callback Routine

```
COMMENT @%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
                        SampCardServicesCallback
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SYNOPSIS:       Callback routine for Card Services events

CALLED BY:      Card Services

PASS:           al      -> function
                cx      -> socket
                dx      -> info
                di      -> 1st word in RegisterClient
                ds      -> dgroup (2nd word in RegisterClient)
                si      -> 3rd word in RegisterClient
                ss:bp   -> MTDRequest
                es:bx   -> buffer
                bx      -> Misc (when no buffer returned)

RETURN:         ax      <- status to return
                carry set on error,
                carry clear on success.
```

```
DESTROYED:      nothing

SIDE EFFECTS:
                None

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SampCardServicesCallback        proc    far
        uses    cx, dx
        .enter

3.4     ; We need to check for the events which can't be included in a linear
        ; sequential jump table

        cmp     al, CSEC_CARD_INSERTION
        je      doInsertion

        cmp     al, CSEC_REGISTRATION_COMPLETE
        jne     handleEvent

        ; We're registered, so we should note this in our driver's state variable

        mov     ds:[amRegistered], TRUE
        jmp     success

        ; Now we handle the other events

handleEvent;

        clr     ah
        mov     di, ax
        shl     di
        cmp     di, endEventRoutineTable - eventRoutineTable
        ja      unsupported

        ; We need to subtract 2 from the value of di since the events are
        ; one-based, not zero-based.

        jmp     cs:[eventRoutineTable][di-2]

        ; For each "routine" mentioned in the table, a label should appear
        ; following this jump
        ;
        ; Example:

doExclusiveReq:
        call    SampCSHandleExclusiveRequest
        jmp     done
```

```
        ;--------------------
doInsertion:
        call    SampHandleInsertion
        jmp     success

doIgnore:

success:
        mov     ax, CSRC_SUCCESS
        clc

done:
        .leave
        ret

        ;--------------------
        ;
        ; The description of what this client supports, when it was created,
        ; etc.
        ;

clientInfo      CSGetClientInfoArgs <
        0,                  ; CSGCIA_maxLen
        size clientInfo,
        mask CSGCIAA_EXCLUSIVE_CARDS or \
                mask CSGCIAA_SHARABLE_CARDS or \
                mask CSGCIAA_MEMORY_CLIENT_DEVICE_DRIVER,
        <                   ; CSGCIA_clientInfo
                0100h,                      ; CSCI_revision
                0201h,                      ; CSCI_csLevel
                <
                        29,                         ; CSDI_YEAR
                        9,                          ; CSDI_MONTH
                        22                          ; CSDI_DAY
                >,                           ; CSCI_revDate
                clientInfoName - clientInfo,  ; CSCI_nameOffset
                length clientInfoName,        ; CSCI_nameLength
                vendorString - clientInfo,    ; CSCI_vStringOffset
                length vendorString           ; CSCI_vStringLength
        >
>
        org     clientInfo.CSGCIA_clientInfo.CSCI_data   ; go back into the
                                                         ; middle of the struct
                                                         ; to place these
                                                         ; strings in the right
                                                         ; place
```


```
clientInfoName char    “Sample PCMCIA Driver”, 0
vendorString    char    “Geoworks”, 0

        ; Your event table should appear here...

SampCardServicesCallback        endp
```

3.4