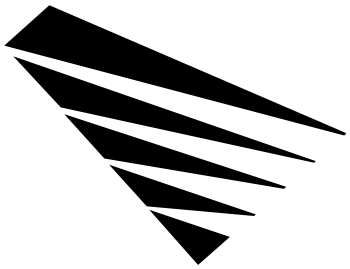

Chapter 1
NLM API Mapper Environment

NIOS in DOS and Windows Environments 2

Re-entrancy in DOS and Windows 3

Detecting Execution Environment 4

Polled LAN Drivers 5

The DOS, MS Windows, and Windows 95 environment-specific NIOS APIs (those beginning with "Dos") have been designed to ease development of API Mapper NLMs which support both the DOS and MS Windows environments. See Figure 1.1 below to review NIOS architecture.

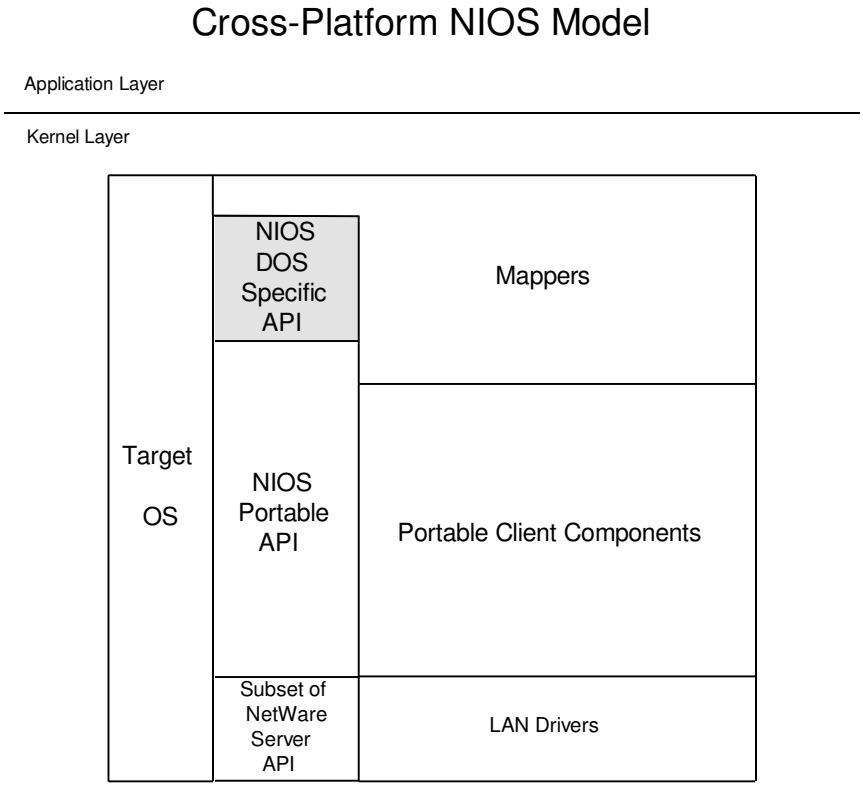

Figure 1.1: NIOS Architecture

NIOS in DOS and Windows Environments

Though the Dos_ NIOS APIs have been designed to be used even while Windows is running, an API Mapper implemented to handle both environments must still take into account some inherent differences between the DOS and MS Windows environments. These issues are elaborated on in the discussion that follows.

In MS Windows, V86 and MS Windows Application (WinApp) code is always executed in the foreground. Ring-0 code cannot invoke V86 or WinApp code while in the context of a hardware interrupt. A Ring-0 interrupt handler that needs to invoke V86 or WinApp code must do so out of a foreground event callback (GlobalEvent).

In a DOS-only environment, V86 code can, as always, be executed directly from the context of a hardware interrupt.

Re-entrancy in DOS and Windows

There are two common cases under DOS and MS Windows where a foreground NLM procedure can be reentered:

1) If the NLM directly or indirectly yields by invoking the NiosPoll function. This case holds for any NIOS-supported environment.

2) If a Mapper NLM directly or indirectly yields by invoking Ring-3 V86 or WinApp code. Both the DOS and Windows VMM functions service foreground events before returning to user mode (V86 or WinApp).

Because hardware interrupts are not serialized (pushed to the foreground) in the DOS-only environment, there exists one other case where an NLM API mapper procedure may be reentered.

3) If the NLM API mapper procedure could be invoked directly or indirectly from a real-mode hardware interrupt handler, then the NLM function could be reentered anytime that NLM enables interrupts.

Note that this case also applies to real-mode API handlers implemented as TSRs; therefore the methods of dealing with this type of reentrancy are the same for TSRs and NLMs: specifically, a queue or a CLI is used to protect global data and/or the function uses stack-based local variables.

Because Ring-0 hardware interrupt handlers are not serialized in either the DOS or Windows environments, an NLM procedure that could be invoked directly or indirectly from a Ring-0 hardware

interrupt handler could obviously be reentered any time it runs with interrupts enabled.

All pertinent DOS environment NIOS services are completely reentrant and therefore can be called from hardware interrupt context. These services are documented as such. However, when the Windows environment is started, many of these NIOS services become non-reentrant to NLM procedures running in the context of a Ring-0 hardware interrupt handler. This is because these DOS NIOS VMM services now map to the appropriate Windows VMM services, which are not reentrant.

Therefore an NLM procedure that wishes to invoke a reentrant DOS NIOS function from the context of a Ring-0 hardware interrupt, while Windows is active, must schedule a foreground event using the *NiosScheduleForegroundEvent* service or the VxD Schedule_Global_Event service. The callback handler would then invoke the needed service.

Detecting Execution Environment

This section describes techniques that allow an NLM binary that supports both the NetWare server and NIOS (e.g. LAN drivers) to determine which of these execution environments is active. This allows an NLM to modify its behaviour at run-time appropriate for the active environment.

Note that an NLM written exclusively for the NIOS environment can simply use the **NiosGetVersion** API function to detect which specific NIOS platform it is executing on (e.g. NetWare server, DOS/Windows 3.1x, Windows 4.x, etc.).

Step 1: Determine if NIOS is present. This is accomplished using the NetWare OS function ImportPublicSymbol and attempting to import the NIOS function **NiosGetVersion**. Note that **ImportPublicSymbol** requires that the function name parameter be length preceded and zero terminated. If the call is unable to locate the public symbol then NIOS is NOT present and therefore the NLM is executing on a NetWare server. For example:

```
UINT32 (*NiosGetVerFuncAddr)( void);

NiosGetVerFuncAddr = (void *)ImportPublicSymbol(
             modHandle,
             "\xE" "NiosGetVersion");

if ( NiosGetVerFuncAddr == NULL)
/* NIOS is NOT present, running on NetWare server */
else
/* NIOS is present, continue to step 2 */
```

Step 2: The second step, if needed, is to determine which particular operating system is hosting the NIOS interface. This is accomplished by invoking the **NiosGetVersion** API function using the address returned in Step 1. For example,

```
NiosVersion = (*NiosGetVerFuncAddr)();
```

Polled LAN Drivers

32-bit NetWare server LAN drivers are now being hosted on a number of desktop operating systems such as DOS, Windows, UnixWare, and NT. Overall this effort has been technically feasible, however one area in which it has not been feasible is support for polled LAN drivers. Most of these desktop operating systems cannot supply a high enough poll frequency needed to provide an adequate level of performance.

There are configurations on the NetWare server where the OS kernel is unable to provide the polling frequency necessary for adequate performance. Because of this Novell recommends in its developer documentation that polled drivers implement and use an interrupt backup mechanism. In the case where the driver's polling function has not been called for a certain amount of time, the adapter will generate an interrupt to the driver, thus allowing the adapter to be serviced. This generally solves the problem on the server since an insufficient polling frequency condition occurs a minority of the time, however in a desktop operating system environment where the polling frequency is insufficient a majority of the time, interrupt backup does not provide an optimal performance solution due to the interrupt time delay (driver/adapter specific) incurred for every adapter event.

The above discussion can be summarized as follows:

1) A polled driver/adapter without an interrupt backup mechanism is not viable in a desktop OS environment. The driver/adapter will function but with extremely poor performance.

2) A polled driver/adapter with an interrupt backup mechanism will function in a desktop OS environment, however performance will fall short of the performance possible using a non-polled driver.

3) A non-polled, purely interrupt driven, driver/adapter provides the best performance in a desktop OS environment. Therefore it is recommended that LAN drivers currently capable of polling be modified to detect the environment in which they are loaded and adjust their behaviour to be either polled with interrupt backup or purely interrupt driven.

Detecting whether or not polling should be used can be accomplished using the **GetPollSupportLevel** function. Note that this API function is NOT available on every platform, therefore the driver cannot import the public directly in its linker definition file, instead the driver should check to see if the public exists during initialization using the **ImportPublicSymbol** function. The entry and exit conditions for **GetPollSupportLevel** are:

```
UINT32 GetPollSupportLevel( void)
```

Returns:	0	Environment does NOT support polling. Polling procedure will never be called.
	1	Limited support for polling. Polling procedure will be called infrequently.
	2	Polling is fully support, however interrupt backup is still recommended.

The following sample code illustrates how to detect the polling support level.

```
UINT32 (*GetPollSupportFunc)( void);

GetPollSupportFunc = (void*)ImportPublicSymbol(
              modHandle,
              "\x12" "GetPollSupportLevel");

if ( GetPollSupportFunc == NULL)
       /* Polling is fully supported */
else
{
       /* Polling may or may not be supported */

       switch( (*GetPollSupportFunc)())
       {
       case 0:
              /* No support for polling. Polling
              Procedure will never be called.  Use
              Interrupt mode */
              break;
```

```
                case 1:
                       /* Limited support for polling.  Polling
                       procedure will be called infrequently.
                       Use Interrupt mode */
                       break;

                case 2:
                       default:
                       /* Polling is fully supported, use
                       Interrupt backup mode */
                       break;
                }
        }
```